Cコンパニオン

Allen I. Holub 著／村上峰子 訳

オーム社

Cコンパニオン

Allen I. Holub 著／村上 峰子 訳

オ ー ム 社

訳者序文

本書はC言語の普通の入門書ではありません．C言語の演算子や制御文についてある程度知っている（またはそれらを書いた本が手近にある）ことが前提になっています．しかし，初心者がCプログラムを書くときに，ミスをしがちな点についてその原因がわかるような知識が多く書かれています．特に次のことが上げられます．

- コンパイラの構成要素とコンパイラはどのように動くか
- C言語が実際に展開されるアセンブラ・プログラム
- Cプログラムの設計の仕方と，プログラムの書式
- 実際的なポインタの使用と多重化した間接アクセス
- 再帰関数
- デバッグの問題と解決法

さらに，洗練されたポインタの使用法や不定数の引数を持つ関数の話題も取り上げられています．このようなプログラムの予備知識を適切に書いている本はほとんどありません．私は，翻訳者としてではなく（初心者の後輩を持つ）中級プログラマとして，本書を翻訳して良かったと思っています．

最後になりましたが，オーム社出版部の方々（度々原稿が遅れてごめんなさい），本書を翻訳する機会を与えてくれた奥田真純氏，中川原亮氏に，深く感謝いたします．

1989年9月

村上 峰子

目　次

1. Cコンパイラ ……1

1.1　コンパイラの各部 ……1

1.1.1　プリプロセッサ ……5

1.1.2　コンパイラ ……7

1.1.3　アセンブラ ……7

1.1.4　リンカ，またはリンク・エディタ ……9

1.1.5　ライブラリアンとライブラリ ……12

1.1.5.1　ライブラリ・メンテナンス ……16

1.2　モジュラ・プログラミング ……19

1.3　他の役立つプログラム ……20

1.3.1　grep ……20

1.3.2　lint ……23

1.3.3　make ……23

1.4　練　習 ……27

2. 2進数演算 ……29

2.1　基数と指数 ……29

2.2　基数変換 ……31

2.3　加算と減算 ……32

2.4　負　数 ……33

2.5　シフト，乗算，除算 ……35

2.5.1　シフト ……35

2.5.2　乗　算 ……36

2.5.3　除　算 ……38

2.5.4　切捨て，オーバーフロー，速度 ……41
2.6　ブール代数 ……42
2.6.1　論理演算子 ……42
2.7　簡単な恒等式 ……44
2.7.0.1　式の反転とド・モルガンの法則 ……44
2.7.0.2　ブール代数の簡単化 ……45
2.8　ビットごとの演算子とマスク ……45
2.8.1　AND，OR，XOR，NOT ……45
2.8.2　マスク ……47
2.9　コンマ，等号，条件演算子 ……48
2.10　バ　グ ……51
2.11　練　習 ……53

3．アセンブリ言語 ……57
3.1　アセンブラとは何か ……58
3.2　メモリ構成とアライメント ……58
3.3　レジスタと計算機の構成 ……60
3.4　アドレッシング・モード ……61
3.4.1　レジスタ直接アドレス ……62
3.4.2　メモリ直接アドレッシング ……62
3.4.3　即値アドレッシング ……63
3.4.4　間接アドレッシング ……63
3.4.5　後置きの自動インクリメント付き間接アドレッシング ……66
3.4.6　自動前置きデクリメント付き間接アドレッシング ……67
3.4.7　指標付きアドレッシング ……69
3.5　命令セット ……71
3.5.1　転送（move）命令 ……71
3.5.2　算術命令 ……71
3.5.3　ジャンプ命令（分岐命令） ……73
3.5.4　サブルーチン・コールとスタック ……74
3.6　ラ ベ ル ……78

3.7　コ メ ン ト …………79
3.8　疑似命令と静的初期化 …………79
3.9　練　　習 …………80

4. コード生成とサブルーチン結合 …………81
4.1　サブルーチンの呼出し …………81
4.2　スタック・フレーム …………90
4.3　サブルーチン呼出しにキャスト（cast）を使う …………96
4.4　返り値（return value） …………98
4.5　シンボル・テーブル（symbol table） …………98
4.6　制御の流れ：if/else，while 等 …………101
4.7　switch …………104
4.8　再配置可能プログラム（relocatable code） …………105
4.9　練　　習 …………106

5. 構造化プログラミングとステップワイズ・リファインメント …………109
5.1　プログラムを書くことは作文である …………110
5.2　プログラム構成 …………119
5.3　コメントのつけ方（commenting）と書式（formatting） …………120
5.3.1　空白（white space）とインデント付け（indenting） …………121
5.3.2　コメントの書き方 …………124
5.3.3　コメントをおく場所 …………125
5.3.4　明白なことを説明しないこと …………126
5.3.5　変数名 …………126
5.4　グローバル変数とアクセス・ルーチン …………127
5.5　移　植　性 …………129
5.6　避けるべきこと …………130
5.6.1　goto の乱用：非構造化プログラミング …………130
5.6.2　#define の誤使用 …………132
5.6.3　switch …………133

5.7 if/else ……133
5.8 Cのスタイル・シート ……134
5.9 練 習 ……136

6. ポインタ ……139
6.1 簡単なポインタ ……140
6.2 ポインタと配列名 ……144
6.3 ポインタの計算 ……149
6.4 角かっこ（[ ]）表記（Square Bracket Notation） ……152
6.5 文字配列；初期化 ……156
6.6 練 習 ……160

7. 高度なポインタ ……163
7.1 複雑な宣言 ……163
7.2 関数ポインタ ……179
7.3 逆の手順 ……186
7.4 typedef ……188
7.5 ハードウエアとやりとりするためのポインタの使用 ……189
7.5.1 絶対メモリ・アドレッシング ……189
7.5.2 CプログラムからIBM PCビデオ・メモリへの直接アクセス ……191
7.5.3 モデル・ハードウエアに構造体を使用する ……193
7.5.4 ハードウエア・インタフェースと移植性 ……194
7.6 練 習 ……195

8. 再帰とコンパイラ・デザイン ……199
8.1 再帰はどのように動くか ……200
8.2 コンパイラの構造 ……205
8.3 文法：コンピュータ言語を表現する ……207
8.4 文法を用いた構文解析 ……209
8.5 再帰的下向き構文解析プログラム ……211
8.6 構文解析プログラムの改善 ……217

8.7　練　　習 ……221

9. Print( ) の構造 ……223

9.1　スタックの再考 ……223
9.2　printf( ) ……224
9.3　doprint( ) で使用されるマクロ ……227
9.4　スタックからの引数のフェッチ ……230
9.5　doprint( )：プログラム ……234
9.6　構造化プログラミングの考え方 ……238
9.7　ltos( ) ……240
9.8　dtos( ) ……242
9.9　練　　習 ……244

10. デバッギング ……247

10.1　デバッグのための printf( ) の使用 ……247
10.2　コメントは入れ子にしない ……250
10.3　lvalue required ……252
10.4　演算子に関連したエラー ……255
10.4.1　優先順位のエラー ……255
10.4.2　評価順序のエラー ……256
10.4.3　誤った演算子の使用 ……258
10.5　制御のながれ ……259
10.5.1　不必要なヌル文，どこにも所属しない { } ……259
10.5.2　誤った if/else の結合 ……261
10.6　マ　ク　ロ ……262
10.6.1　マクロの優先順位の問題 ……262
10.6.2　期待に反するマクロ引数の置換え ……263
10.6.3　マクロの副作用 ……264
10.7　16 ビットではない int ……264
10.7.1　精　度 ……264
10.7.2　マクロでの不完全な文 ……265

10.7.3 マスク ……265
10.7.4 定数は int である ……266
10.8 自動型変換の問題 ……266
10.9 extern 文の欠如，または暗黙の extern 文 ……269
10.10 I/O ……271
10.10.1 scanf( ) ……271
10.10.2 getc( ) はバッファを持った関数である ……272
10.10.3 変換される I/O 対変換されない I/O ……273
10.11 演算子を除く最適化プログラム ……274
10.12 練 習 ……276

参考文献 ……277

付録 A：優先順位図 ……285
付録 B：シェル・ソート ……287

索 引 ……291

はじめに

C 言語のテキストは，たいていとても教養ある読者向けか，または，まだ未熟な読者向けに書かれている．どちらのタイプのテキストも共通の問題を持っている．それは，技術的な予備知識となる説明を欠いているということである．初級用のテキストでは，読者はそのような説明はわからないと想定しているし，上級用のテキストでは読者はすでに必要な知識を持っているものとしている．

中間レベルのプログラマにとって，基礎的なテキストは退屈で，十分に広範囲な内容ではない．それはプログラマにとって真に必要なことを載せていない．

初心者のCプログラマにとっては，さらに困難な問題がある．C 言語が，どのように動作するのかをほんとうに理解する前に，たくさんの予備知識が必要となる．初心者はさまざまな演算子が何をするのか，ましてそれらがどう使われるのかもわからない．それで，予備知識なしで言語を学ぼうとして失敗している．また，コンピュータの下部構造がわからなければ，C プログラムをデバッグすることもたいへん難しい．このような予備知識を与えていなければ，”初心者のためのC 言語”のようなテキストは，実際には読者を欺いていることになる．必要条件なしで言語を学ぶことはできない．

本書では，2つの方法でこの問題を解決している．C 言語をマスターすることと，カーニハンとリッチー（参考文献参照）が書いたような典型的なテキストを理解すること双方に役立つ予備知識を述べている．また，ポインタの洗練された使いかたや，可変数の引数を持ったサブルーチンのような上級向けの話題についても詳しく取り上げている．

本書の使いかた

本書からはCの初歩は学べない．実際には，読者は本書を使いながら一般のテキストを参考に勉強するはずである．次の2つのうち，ひとつの方法で本書を使うことができる．1章から順に，本書と一般の本の章とを交互に読む．あるいは，多少なりともでたらめな章順に読む．もっとも基本的な章でさえ読者が言語について（例えば変数の宣言方法や，whileループとは何か等）少しは知っていると仮定している．

本書は4つのセクションに分けられる．1～5章は技術的な予備知識を扱っている．6章はポインタの概論であり，この題材はたいていのテキストに含まれている．しかし，異なる視点から再び取り上げるのは有意義であろう．7～9章はC言語の応用に発展している．これは読者の知識を磨くのに役立つだろう．これらの章では，C言語の完全な知識が要求される．最後に，10章では一般的なデバッグの問題について扱っている．各章は次のような構成になっている．

1. コンパイラの使用

1章では，Cコンパイラの各部とそれらが互いにどう動くかを討議する．プリプロセッサ，コンパイラ，リンカ，ライブラリアン等は，モジュラ・プログラミングの基本的概念としてすべて説明されている．この章では，Cプログラマに役立つmakeのようないくつかの補助プログラムを取り上げている．

2. 基本原則：2進数演算

ここでは，2進数演算の基本原則を討議する．討議には，種々の基数，ある基数から別の基数への変換のしかた，計算機で負数をどう表すか，乗算と除算のしかた，単純なブール代数などを含む．そして，計算機での計算方法によってCプログラムに生じる問題に重点をおいている．

3.　C プログラマのためのアセンブリ言語プログラミング

3 章では，あるやりかたでアセンブリ言語について説明する．それは，C での処理の仕方を読者が理解する助けになるだろう．種々のアドレッシング・モードと簡単なオペレーション，また，同時にメモリ・アライメントについても述べている．

4.　コード生成とサブルーチンの結合

4 章では，C コンパイラが，通常，生成するコードを討議する．また，サブルーチンがどのように呼ばれるかについても詳述する．C プログラムにおけるささいでみつけるのが難しいバグの多くは，誤ったサブルーチン結合の結果である．この章は，そのようなバグをより簡単に発見する助けとなるだろう．ただし，読者は，4 章を読む前に 3 章の内容を理解しておくべきである．

5.　モジュラ・プログラミング

5 章では，大きな C プログラムの設計と書きかたについて述べる．また，フォーマット規則についても同様に述べる．

6.　ポ　イ　ン　タ

ここでは，ポインタの基本原理が詳しく討議される．ポインタは，C 言語において，もっとも理解しにくい部分である．そして，たいていのテキストでは十分に扱っていない．題材は実際の面から扱われている．この章では，メイルボックスのようなものに頼るのでなく，むしろ種々のポインタが計算機の中で実際にどう動いているかを説明する．読者はこの章を読む前に 3 章の内容を理解しておくべきである．

7.　高度なポインタ

7章は，6章の終わりの部分からの続きである．ポインタの複雑な使用，多重レベルの間接，ポインタと配列を使った複合型等について詳しく調べる．また，複合型の宣言文もここで詳しく表している．この章ではハードウェア・インタフェースについても討議される(例えば，IBM-PCのメモリマップ・ディスプレイにアセンブリ言語を使わずに直接働きかける方法を討議する)．

8.　再帰呼出しとコンパイラ理論

ここでは，再帰呼出しについて述べる．再帰呼出しのサブルーチンが繰り返し呼ばれているときに，計算機内で実際に何が行われているのかをみることによって，再帰呼出しがどのように動作するのかを詳しく調べる．小さなコンパイラのような語句解析プログラムを解析して，再帰呼出しの実際的な使用もみてみる．読者は，この章を読む前に4章と6章の内容を理解しておくべきである．

9.　printf( )：可変数の引数を持ったサブルーチン

ここでは，print( )の変形版をみて，Cの複雑なアプリケーションを検討する．読者は，この章を始める前に7章と8章を理解しておく必要がある．

10.　デバッギング

10章では，さまざまなデバッギング上の問題を検討する．そして，これらの問題の解決法について述べる．

本書のCプログラムは，小さな例題やポインタの章で，やや奇妙に思える資料を含んでいるが，すべてIBM PC/AT上でMicrosoft C Compiler version 3.0を使用してテストされている．これらのプログラムは，Lattice version 3.x，またはUNIXで問題なく走るはずである．他のコンパイラについては保証できないが，それが標準に準拠しているならば問題はないはずである．

謝辞

本書の最初と最後の原稿に貴重な意見を与えてくれた，友人であり隣人でもある Bill Wong に感謝したい．わたしが注意してみるより多くのバグと誤植を本書に発見した幾人かの生徒，特に Tom Clement に感謝する．Dale Coleman は，正式にはアルバニーの Computerland で，親切にもお店のレーザプリンタで527ページの原稿をプリントアウトさせてくれた．このことによって，わたしは同い年のデイジーホイールのプリンタがガッガッと進むのを見ずにすんだ．最後に，Brian Krenighan と Steve Wampler は，最終原稿にたくさんの有効なコメントと批評を与えてくれた．彼らが注意深く再検討してくださったことをありがたく思っている．

1 Cコンパイラ

本章では，Cコンパイラがどのように構成されているか，そして，一般にそれがどう使われているかを紹介する．また，Cプログラマに役立つ，Cコンパイラ以外のいくつかの補助プログラム（grep や make のような）も紹介する．すべてのCコンパイラは構造上は同様だが，実際にプログラムをコンパイルするやり方はコンパイラによってまったく異なる．読者は自分のコンパイラの使用方法を知るために，コンパイラのドキュメンテーションを調べるべきである．本章では，コンパイラが共通して持っている部分とその部分がどう使われるかを検討する．なお，本書では UNIX とマイクロソフト社のコンパイラを使用する．

1.1 コンパイラの各部

Cコンパイラは通常ひとつ，または，協調して動く複数のプログラムからなる．コンパイレイション自体はいくつかのフェーズ（phase）に分割され，各フェーズはひとつ，または複数のパスに含まれている．あるフェーズは，実際のコンパイルが始まる前に行われなければならない．例えば，コンパイラのプリプロセッサ・フェーズは，ソース・コードからすべてのコメント文を取り除き，そして，すべてのマクロ（**#define** や **#ifdef** 等）を置き換える．プリプロセッサ・フェーズの出力は，コンパイラの字句解析フェーズに渡される．字句解析フェーズは，入力された文字を他のフェーズが使いやすい内部表現に変換する．一方，パスは，入力ファイルの読込みと出力ファイルの生成を含んでいる．コンパイラの第1パス（パス1）はソース・ファイルを読んで，一時的な出力ファイルを生成する．パス2はこの一時的なファイル（テンポラリ・ファイル）を読む，そして，代わり

にパス2の出力ファイルを生成する．処理は，実行可能プログラムができるまで続く．コンパイラのいくつかのフェーズは，しばしば結合されてひとつのパスになる．たいていのコンパイラは，少なくとも3つのパスからなる．マイクロソフト社のCコンパイラは，実際4パスである（4つのプログラムp0.exe，p1.exe，p2.exe，p3.exeがある）．

コンパイラのフェーズは，しばしばドライバ・プログラムによって隠されている．UNIXプログラムのccや，マイクロソフト社のclとmscがその例である．ドライバは，他のプログラムをまるでサブルーチンのように走らせるプログラムである．サブプログラムは，テンポラリ・ファイルを通して互いに通信する．そして，ドライバは，コマンドを使って他のプログラムと通信する．mscドライバは，p0.exe, p1.exe, … と順序良く実行する．ドライバは，テンポラリ・ファイルがもはや必要なくなったときにそれを削除する．そして，同様にそのほかのハウスキーピング(訳注1)も削除する．たとえ，パスの動きが隠されているとしても，さまざまなパスが何をしているか知ることは有益である．コンパイラを構成しているプログラムを図1·1に示す．

Microsoft	UNIX	機能
cl	cc	ドライバ・プログラム
p0	cpp	Cプリプロセッサ（パス1）
p1	c0	Cコンパイラ　パス2
p2	c1	Cコンパイラ　パス3
p3	c2	Cオプティマイザ（最適化　パス4）
masm	as	アセンブラ
link	ld	リンカ
lib	ar	ライブラリアン

図1·1 コンパイラのパス

UNIXコンパイラccのいくつかのバージョンは，コマンドに -v を規定すれば，これらすべてのプログラムが実行されるときにその名前を表示する．図1·2はコンパイラのさまざまなフェーズ間の関係を示す．それぞれのパスによって生成されたテンポラリ・ファイルも示されている．

UNIXのccドライバをみることによって，ドライバがどのように使われているかを示そう（マイクロソフトのclドライバは，ccにとても似ている）．ccのコマ

訳注1：一般には「問題の解に関与しないプログラム内のルーチン．入出力装置に必要な予備的操作ルーチン．」

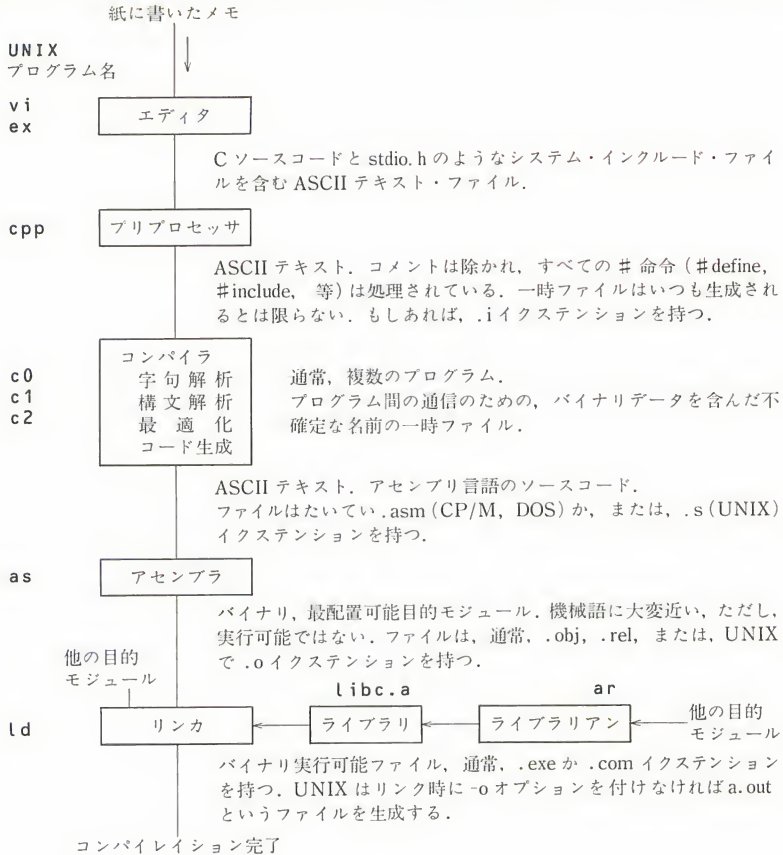



図 1・2 Cコンパイラのさまざまなパス

ンドは

```
cc [options] files...
```

である. ここで, files はファイル名である. ファイル名が .c で終わっていれば, そのファイルを cc は C プログラムであるとする. 同様に, ファイル名の終わりが .s ならばアセンブリ・プログラム, .o ならばオブジェクト・モジュールであるとする. ファイル名のイクステンション (.c, .s, または .o) に基づいて, cc はコンパイル, アセンブル, または, さまざまなファイルを適切にリンクする (これらの動作は, あとで詳しく説明する.). 次のコマンドを例に説明する.

```
cc hickory.c dickory.s doc.o
```

hickory.c はコンパイルされ（cpp，c0，c1 を使用する），アセンブルされる（as を使用する）．dickory.s はアセンブルだけされる（as を使用する）．これらの処理で，hickory.o と dickory.o が生成される．次に，hickory.o，dickory.o と doc.o が同時にリンクされ（1d を使用する）．さらに，スタンダード・ライブラリ（libc.a と呼ぶ）が自動的にプログラムにリンクされる．最後に，生成されたテンポラリ・ファイル（hickory.o，dickory.o を含む）が cc によってすべて削除される．

cc は，コマンドにいくつかのスイッチを使う．さらに，cc は自身に関係のないスイッチをリンカ（1d）に渡す．次に cc のより有用ないくつかのオプションを記す．

-c　　コンパイルはするが，リンクはしない．コマンドに規定した各 .c または .s のファイルに対応する目的モジュール（.o イクステンションがついている）が生成される．しかし，これらの目的モジュールはリンクはされない．このオプションは makefile を使うときに役立つ（このことについては簡単に後述する）．

-Dname　　name に対する #define 文をつくる．これはおもにデバッグに使われる．例えば，デバッグ用の部分を

```
#ifdef DEBUG
    printf("This is a debug diagnostic");
#endif
```

のように #ifdef，#endif で囲むことができる．もしプログラムを

```
cc -DDEBUG prog.c
```

でコンパイルすれば，printf（ ）文はコンパイルされるだろう．また，マクロに値を割り当てることもできる．

```
cc -DPROCESSOR=8086 prog.c
```

はプログラムに次の文をつくる．

```
#define PROCESSOR 8086
```

-O　　最適化をするパス（cc2）を呼びだす．コマンドで -O を規定しなければコードは最適化されない．-O が規定されるとコンパイルに時間がかかるため，デバッグ作業が完了するまで通常は

使用されない．このオプションは次のオプション -o name と混同すべきではない．

-o name　　この -o name を指定しなければ，a.out にリンクされることになる．しかし，例えば

```
cc -o reindeer dasher.c dancer.c donner.c blitzen.c
```

は最終的に，a.out ではなく，reindeer にリンクする．

-s　　アセンブリ言語のファイルを生成し，そして終了する．そのファイルは .s イクステンション（マイクロソフト社を使っていれば，**.asm**）を持つ．

1.1.1 プリプロセッサ

プリプロセッサは，エディタでつくられたアスキー形式のテキストを入力とする．慣習では，C 言語のソースファイル名は .c イクステンション（program.c のように）で終わり，インクルード・ファイル（#**include** 機能でプログラムに組み込む）はそのファイル名の最後に .h を持つ．

プリプロセッサは，2 種類の機能を持っている．ひとつは，ソースコードからすべてのコメントを取り除くこと，そして，もうひとつはプリプロセッサへのすべての命令を処理することである．C ではこれらの命令はすべて最初の文字が # である．ほとんどのコンパイラによってサポートされているプリプロセッサ命令が，図 1·3 に示されている．互換性を保つために，# はつねにもっとも左のカラムにあるべきであり，# に続いてどんな空白（ブランク，タブ，改行）もおくべきではない（たとえ読者のコンパイラが，空白をいれてあってもかまわないとしても）．

#define	–	マクロの定義，識別子を文字列と置換する
#undef	–	マクロの取り消し
#include	–	この命令のある行をファイルの内容と置換する
#ifdef	–	マクロが先に #define で定義されていればコンパイルする
#if	–	式が真ならばコンパイルする
#else	–	先行する #if の式が真でない，または，#ifdef の識別子が未定義のときコンパイルする
#endif	–	#if，#ifdef，#else の範囲の終わり

図 1·3　共通のプリプロセッサ命令

すべてのプリプロセッサ命令は，ある種の文字列の置き換えを含んでいる．例

えば，マクロの呼出しは，さきに ＃define 命令で定義された文字列が置き換えられる．＃include 命令は，命令の一部で定義されたファイルの内容と置き換わる．また，＃ifdef 命令は，文字列を削除する（ヌル文字列で置き換える）こともできる等である．

プリプロセッサは，コンパイラの一部ではない．プリプロセッサは，C の構文や優先順位を理解しない（何が識別子かは知っているけれども）．これがわからないために，いくつかのバグが起こりうる．例えば

```
#define PRINTS(s)       printf("string is <%s>", s);
```

が

```
PRINTS( buf );
```

で呼ばれると，次のように展開される．

```
printf("string is <%buf>", buf);
```

同様に

```
#define  SQUARE(x)      (x * x)
```

が

```
y = SQUARE( var++ );
```

で呼ばれたとき，次のように展開される（var が 2 回インクリメントされる）．

```
y = (var++ * var++);
```

3 番目の例は，SQUARE が

```
y = SQUARE(x + y);
```

で呼ばれたとき，次のように展開する．

```
y = (x + y * x + y)
```

演算子＊は演算子＋より優先度が高いので，この式は次のように評価される．

```
y = ( x + (y * x) + y)
```

これは，期待どおりではない．この最後の問題は，SQUARE を

```
#define  SQUARE(x)      ((x) * (x))
```

で再定義することにより解決できる．マクロを使うときは，よく注意すべきである．

多くのコンパイラは，プリプロセッサが処理した後も，ソースコードでの位置がわかるようにするオプションがある．これは，複雑なマクロを使用しているときのデバッグにも有効である．例えば，cc と cl はどちらもコマンドに -P があれ

ば，プリプロセッサの出力ファイルが生成される（この出力ファイルはどちらのコンパイラも .i イクステンションを持っている）．UNIX プリプロセッサ（ccp）は，完全に独立したプログラムなので，プリプロセッサだけを走らせることができる．ccp は /lib ディレクトリにあるので注意すること．もし /lib が自分のサーチ・パスになければ，ccp プログラムを走らせるために，次に記すフル・パスネームを与えなければならないだろう．

```
/lib/cpp prog.c prog.i
```

1.1.2 コンパイラ

コンパイレイションの次の段階は少なくとも 4 つのフェーズを含んでいる．それらは字句解析，構文解析，最適化，コード生成である（字句解析と構文解析は 8 章で詳述する）．最初の 2 つのフェーズはその間に，一般的なアセンブリ・コードのような中間コードからなるファイルを生成する．

最適化は中間コードを入力とし，そして，その所要記憶容量を小さくし，より速く走らせるようにする．たいてい，この両方の長所を同時に得ることはできない．コードを小さくするか，速く実行するか，どちらか一方しかできない．cc は，コマンドに -O があるときだけ最適化を行う．そのため，プログラムのデバッグをしているときに，最適化に時間を費やさなくてすむ．最適化は，予想できないやり方でコードを再配置する．したがって，アセンブリ言語のデバッガ（symdeb や adb）を使うのなら最適化はしないほうがよい．

最適化は，順序をばらばらに変えられた中間コードを出力する．コード生成のパスは，この中間コードをアセンブリ言語に翻訳する．コード生成以外のすべてのコンパイレイションを行うことが，汎用のフロントエンドにすることを可能にする．異なるコード生成プログラムに置き換えることによって，コンパイラの他の部分の修正をせず，さまざまな計算機にあった目的コードをつくることができる．

4 つのフェーズすべてが，ひとつのパスで行われることがある．けれども，多くのものは，字句解析と構文解析がひとつのプログラムによって行われ，最適化は第 2 の，そして，コード生成は第 3 のプログラムで行われる．

1.1.3 アセンブラ

アセンブラは，アセンブラ言語を入力とし，実行可能コードにとても近いもの

を出力として生成する．アセンブリ言語の中間ファイルをつくらないですむようにコード生成フェーズに組み込まれているアセンブラもある．しかし，アセンブリ言語のプログラムも出力することができる機能を持っているものが多い．(cc と cl ではコマンドに -S オプションをつけてコンパイルできる)．多くのコンパイラは，アセンブリ言語の出力に，コメントとしてC言語のソースコードを挿入できる．

コンパイラによってつくられたアセンブリ言語をみることができるのは，とても重要なことである．adb，ddt，symdeb，または，debug のようなデバッガを使用するためには，アセンブリ言語を必要とする．さらに，特に効率について関心があるならば，コンパイラがハイレベルの構造で処理した内容を厳密にみることは有益だろう．コンパイラには，多くのほかのプログラムと同様に，バグがある．これは，8086 ラージモデルのコンパイレイションや，いくつかの新しいコンパイラについては特にそうである．もしアセンブリコードをみることができれば，コンパイラが正しいコードを生成したかどうかわかる（C をアセンブリ言語に関係付けるのは容易である．しかし Pascal，LISP，APL のような言語では容易ではない）．アセンブリ言語をみることの最大の理由は，「手」による最適化である．最初からアセンブリ言語で書くより，C でプログラムを書き，コンパイルし，それから計算機で生成されたアセンブリ言語を手で最適化するほうがたやすい．

UNIX のアセンブラ (as) とマイクロソフト社のアセンブラ (masm) は独立したプログラムであり，それだけで走らせることができる．慣例では，UNIX のアセンブリ・ソース・ファイルは .s のイクステンションがあり，MS-DOS のファイルは .asm がある．

アセンブラの出力は，再配置可能な目的モジュールである（しばしば，単にモジュールと呼ばれる）．Pascal とは違い，C プログラムは通常，いくつかのファイルに分けて書かれる．これらのソースファイルは別々にコンパイルされ，コンパイレイションの最後の段階（次にみるリンカによって）でつなぎ合わされる．あるファイルに存在するサブルーチンは，第2のファイルに存在する，ほかのサブルーチンを呼ぶことができる．しかし，コンパイラは，第1のサブルーチンをコンパイルしているときに，第2のサブルーチンがメモリのどこにあるかを知ることはできない（なぜなら，2つは異なるファイルにあるからである）．同じ状況が，グローバル変数でも起こる．すなわち，あるファイルで定義されたサブルー

チンが，他のファイルで定義されたグローバル変数を使うことがありうる．しかし，コンパイラはグローバル変数が存在するところを知る方法がない．このような，他のファイルにあるサブルーチンや変数は，外部オブジェクトと呼ばれる．

アセンブラは，できるところまでは機械語に翻訳しようとする．それは，外部オブジェクトがメモリのどこにあるのかわからないからである．しかしながら，仮のアドレスで，外部オブジェクトへのすべての参照を置き換える．それから，それらの仮のアドレスの位置とおのおのの仮のアドレスに関連する変数の名前をリンカに知らせる表をつくる．

再配置可能な目的モジュールは，外部オブジェクトにアクセスする前にパッチされなければならないから，実行可能ではない．リンカに通した後で実行させなければならない．

1.1.4 リンカ，またはリンク・エディタ

リンカは，さきに述べたコンパイレイション処理の結果できた再配置可能な目的モジュールの表を入力とする．次に，それらのモジュールをひとつのプログラムにつなぎ合わせる．そして，すべての仮のアドレスを本当のアドレスで置き換える．例えば，コマンド

```
cc -c larry.c
cc -c curly.c
cc -c moe.c
```

は，ファイル larry. c，curly. c，moe. c にある C のソースコードをコンパイルおよびアセンブルし，3つの目的モジュール larry. o，curly. o，moe. o をつくる．それから，それらのファイルは，リンカを使ってひとつの実行可能プログラムに変えられる．すなわち

```
ld -o stooges larry.o curly.o moe.o -lc
```

である．別のやり方は

```
cc -o stooges larry.o curly.o moe.o
```

である．ld は，stooges という実行可能プログラムを生成する（-o だから）．-lc は，コマンドに与えられた3つのモジュールを printf( ) のような標準 I/O ルーチンを含んでいる標準ライブラリ /lib/libc. a とリンクすることを ld に知らせる．-lc は ld ではなく cc を使うときには必要ない．マイクロソフト社の cl ドライバは cc が使用されたのと同様に使うことができる．マイクロソフト社のリンカは

```
link larry.o curly.o moe.o,stooges.exe,,libc.lib
```

で libc. lib を明示して呼び出される．ここで，コンマで区切られた第2の引数が出力ファイル名である．もし，第3の引数があればリンクマップがつくられるだろう（リンクマップは，すべての外部オブジェクトとそれらのメモリ上の実際の位置の表である）．最後の引数は，プログラムの終わりにリンクされるライブラリの名前である．

リンカの機能をより詳しく説明する前に，専門的な説明をしなければならない．C（適切には，カーニハンとリッチーの場合）は，変数やサブルーチンの宣言と定義を区別している．宣言という言葉は告知を意味している．それで，変数宣言はコンパイラに変数の存在を知らせている．また，宣言では記憶を割り当てず，ただ，コンパイラに変数がどこかに存在することを知らせるだけである．コンパイラは，リンク時にリンカが変数のありかをみつけることを想定している．変数定義は，実際に変数に対して記憶を割り当てる．つまり実際に変数をつくる．

このことば—定義と宣言—の選択は適切でない．なぜなら，たいていの文献では，宣言ということばは変数に割り当てられている記憶の場所を参照するのに使われている．これは，カーニハンとリッチーが使っていることばの意味とは反対である．結論として，私は本書では一貫してより広く受け入れられている意味で，2つのことばを使用してきた（2つのことばを区別していない．すなわち，2つのことばは，カーニハンとリッチーがいうところの定義と同じ意味で使われる）．必要があれば，私は，定義と宣言の代わりに，割当て（allocation）と参照（reference）を使う．

しかしながら，この区別は重要である．ひとつの変数に対する記憶は，ただひとつの場所にだけ割り当てられる．けれども，変数は，それを必要とする多くのファイルから参照することができる．C では，参照は予約語 **extern** でつくられる．すなわち，**extern** は，名前付き変数（または，サブルーチン）が他のファイルに定義されていることをコンパイラに知らせる．しかし，変数がまるで現在のファイルに定義されているように使用可能であることも意味している（リンカが変数の実際の位置をみつけるだろう）．**extern** で宣言されている変数には記憶は割り当てられない．もし **extern** がなければ，そのとき記憶が変数に対して割り当てられる（理論的には，コンパイラがリンカに割当てをさせることもある．これについてはのちほど説明する）．特に，外部サブルーチン（変数ではない）は，使うだ

けで暗黙のうちに外部サブルーチンであると宣言することができる．この場合，そのサブルーチンは**int**を返すものとされる．

リンカの仕事は，すべてのサブルーチンと変数の参照を，それらの宣言と結び付けることである．さらに，リンカが変数の宣言をみつけられなければ（つまり，参照しかなければ），通常は変数に対して記憶を割り当てる．これは，リンカに記憶を割り当てさせるコンパイラに問題を起こすかもしれない（予約語**extern**がないときに，コンパイラ自身が記憶割当てを行うよりも）．このコンパイラは，同じ名前を持った2つの静的でないグローバル変数を，同じ変数であると想定する．実に頭の痛い問題である．後の章でこの問題を回避する方法を検討する．

リンカは，つなぎ合わせた目的モジュールを，次の2つのうちひとつの形にすることができる．アセンブラからの出力のような，ファイル別に独立した目的モジュールか，または，ライブラリ（library）である．ライブラリは，ひとつのファイル（通常，名前に .lib イクステンションを持っている）に合併された目的モジュールの集合である．ファイル内のモジュールは個々に独立している．しかし，リンカのコマンドには，ひとつのファイルにおかれているので，必要なモジュール名を長く連ねないでひとつのファイル名を書く．

ひとつの目的モジュールを分けることはできない．つまり，ライブラリが走査されるときに，もし特定のモジュール内のひとつのサブルーチンが必要ならば，たとえ同じモジュール内の他のサブルーチンが必要でなくとも，モジュール全体が最終的なファイルに出力される．最初に，ひとつのソースコードにあったすべてのサブルーチンと，広域データはひとつの目的モジュールになる．このように，ソースコードのレベルでは，ひとつのモジュールはひとつのファイルと同じである．このソースコードとファイルの関係は，ライブラリに対しては保持されない．ライブラリは，複数の目的モジュールを含むひとつのファイルだからである．ひとつのモジュールは，単一のファイルで始まったひとかたまりのコードである．しかし，ライブラリは複数のモジュールを含む単一ファイルである．モジュールとファイルはこのレベルでは同一ではない．

リンカは，ライブラリの必要とされる（参照される）モジュールだけを最終的なプログラムに組み込む．一方で，ライブラリに含まれていない独立した目的モジュールがコマンド上にあれば，それは実際に使用されなくとも，必ず最終的なプログラムに入っている．

リンカを使用することの真の利点は，いったんプログラム修正が終わると，修正したファイルだけを再びコンパイルすればよいことである（そのプログラムにはまだ他のファイルに目的モジュールがあるとする）．それから，新しい目的モジュールを使ってプログラムを再びリンクすることができる．

1.1.5 ライブラリアンとライブラリ

厳密にいえば，ライブラリアンはコンパイラの一部ではない．ライブラリをつくり，維持するためにだけ使用される独立したプログラムである．たいていのライブラリアンは，ライブラリをつくり，ライブラリにモジュールを加え，ライブラリからモジュールを削除し，ライブラリにあるモジュールを同じ名前の新しいモジュールで置き換えることができる．ライブラリから，ひとつのモジュールを抜き取れるライブラリアンもある（ライブラリ・ファイルからモジュールを取り除き，自動的にひとつの .o，または .obj ファイルにおく．それはまるで最初からライブラリにはおかれていなかったかのようである）．

ライブラリにあるモジュールは，そのモジュール内のサブルーチンやデータが使用されていなければ，最終的なプログラムには含まれない．そのため，ライブラリは通常かなり大きい．ライブラリは，複数のサブルーチンを含むモジュールが 100 個以上あることもしばしばある．例えば，読者のコンパイラの資料に記述されているライブラリ・ルーチンは，すべて個別のモジュールであり，かつ，すべて同じライブラリ・ファイルの一部分である．

多くのリンカは，ライブラリをはじめから終わりまで 1 度走査するだけである．したがって，モジュールがライブラリに挿入される順番がしばしば重要になる．リンカは，すべての未解決外部変数の表を管理する．走査している間に，その表にある外部参照と同じ名前を持つサブルーチンがみつかれば，リンカはそのサブルーチンを含んでいるモジュールを最終的なプログラムに挿入する．新しく挿入されたモジュールに外部サブルーチン参照があれば，この新しい参照をリンカの表に加える．そして，ライブラリ内のサブルーチンがやはりライブラリ内にあるサブルーチンを呼んでいるならば，呼びだされるサブルーチンを含んでいるモジュールが，そのサブルーチンを呼ぶモジュールの後になければならない．呼ぶほうのサブルーチンが挿入された後に，呼ばれるほうのサブルーチンはライブラリにおかれていなければならない．モジュール内のサブルーチンは，すべて最終的なプ

ログラムの一部になるから，モジュール内のサブルーチンの順番は重要ではない．特に，でたらめな順番のライブラリでも受け入れるリンカは，きれいに並んだライブラリを与えられるとより速く動作するだろう．

ライブラリに入れられるモジュールでもうひとつ考慮することは，モジュール内のサブルーチンの数である．ライブラリ・モジュールは，通常外部から参照されることが可能なサブルーチンをひとつだけ持つことができる（**static** で宣言されたサブルーチンは，外部からの参照は不可能である）．モジュール内の他のサブルーチンは，ひとつの外部ルーチンを補助するはずである．モジュールを小さくつくることによって，最終的なプログラムで使われないサブルーチンをリンクせずに，プログラムのサイズを小さくする．

リンカがひとつのプログラムを編集するとき，サブルーチンの能力がすべて実際に使用されているかどうか知る方法はない．サブルーチンに対する呼出しがすでにリンクされているコードにあるかどうかということだけがわかる．これは，汎用の出力関数 fprintf( ) のようなハイレベルの関数を使用するときに問題を生じる．fprintf( ) はスクリーンへも，ファイルへも，どのような MS-DOS のデバイスにも書くことができる．したがって，stderr にエラーメッセージを書くためだけに fprintf( ) を使っているのかもしれないのに，仮にそれを使えば，最終的なプログラムは，ほとんどディスク・インターフェース・ライブラリになってしまう．リンカは fprintf( ) のディスクに関係ある機能の中の使わない部分がわからない．つまり，fprintf( ) でディスクに書くということだけがわかる．同様に，fprintf( ) は浮動小数を書くことができる．したがって，たとえ整数を書くために fprintf( ) を使うだけでも，浮動小数計算のライブラリもすべて最終的なプログラムに入ってしまう．

この問題には２つの解決策がある．汎用の関数を使わないようにするか，または，自分用の fprintf( ) の変形版を書くことである．最初の方法は汎用のルーチンの小さな部分を使うときには実用的である．つまり，文字列を書くためだけに fprintf( ) を使っているならば，代わりに fputs( ) を使うことができる．自分用の fprintf( ) の変形版を書くという第２の解法は，長い目で見ればより実用的である．多くのコンパイラの製造業者はほとんど独自の，いくつかのライブラリをコンパイラとともに供給している．これらのライブラリのひとつは浮動小数に対処できない fprintf( ) の変形版を含んでいるだろう．別のライブラリは，ディス

クには書けない fprintf( ) の変形版を含んでいるだろう．リンク時に適当なライブラリを選ぶことによって，最終的なプログラムに入れる fprintf( ) の版を選択することができる．

もうひとつ関連した問題がある．コマンド上に明示して，リンカに与えられたすべての目的モジュール（ライブラリの一部ではない）は，最終的なプログラムに入れられる．リンカがライブラリを走査するとき，まだ，ないサブルーチンだけを探す．自分用のライブラリ・ルーチン（もとのライブラリ・ルーチンと同じ名前である）をつくり，リンカのコマンドにそのルーチンを含んでいるモジュール名を書くならば，自分でつくったルーチンが最終的なプログラムの一部とされる．そのとき，リンカはライブラリにある同じ名前のサブルーチンを探そうとはしない．ここで，問題は，ライブラリ内にある他のサブルーチンが，置き換えられたサブルーチンを呼んでいるかも知れないことである．呼出しの点では，自分のルーチンを標準のライブラリ・ルーチンに正確に似せるよう特に注意しなければならない．つまり，自分のルーチンは通常のライブラリ・ルーチンと同じ引数を持たせ，同じ値を返すようにする．たとえ戻り値や引数をひとつも使わないとしてもである．

コンパイラのメーカーから供給されている標準のライブラリは，たとえ標準の I/O ライブラリのルーチンは使用しないとしても，最終的なプログラムにリンクされなければならない．コンパイラは，2〜3 の小さなサブルーチンが，最終的なプログラムにあり，それらのサブルーチンが標準ライブラリのルーチンを呼ぶことを想定している．このルーチンの集まりは，実行時ライブラリと呼ばれ，通常，I/O ルーチンとともに標準ライブラリ・ファイルにおかれている．実行時ライブラリ・ルーチンは，倍精度計算や switch 文の処理などを行う．

特に興味をひくひとつの実行時サブルーチンは，root または start-up モジュールである．root モジュールはオペレーティング・システムから制御を受け取るサブルーチンである．main( ) サブルーチンは他のサブルーチンのように，root モジュールから呼ばれる（main( ) は，他と同様に，単なるサブルーチンである．main( ) 関数を必要とするただひとつの理由は，root モジュールがそれを呼ぶからである．だから，リンカは main( ) の存在を要求する）．root モジュールは I/O ライブラリで使用されているさまざまなグローバル変数を初期化する．通常，コマンド・ラインから受けとった argv 配列（注 1）を組み立てる．CP/M や初期

注 1：コマンドの引数が入っている文字列へのポインタの配列．

のMS-DOSのようなオペレーティング・システムでは，入出力切換えもrootモジュールで行われる．

rootモジュールの使用は，プログラムをできるだけ小さくしようとするときにも利益がある．切換えはディスクを使うから，明示して切換えをするならば，rootモジュールは全体のディスクI/Oシステムを呼ぶだろう．切換えを行わないようにrootモジュールを書き換えることによって，切換えを阻むことができる（この処理の例は参考文献にある）．rootモジュールと同じ名前のサブルーチンを書いて，自分のプログラムにそのサブルーチンをリンクすることができる．main( )はrootモジュールに呼ばれるので，自分でrootモジュールを書くのならばmain( )サブルーチンは必要ではない．

UNIXでは，標準ライブラリ（/lib/libc.a）が自動的にccによってプログラムにリンクされる．マイクロソフト社のリンカは，コンパイラがリンカに対して，目的モジュールに，標準ライブラリを使用するための情報を入れるなら，リンクする．けれども，すべてのコンパイラが，この情報を入れるわけではない（ここに書いたように，マイクロソフト社のCコンパイラは入れるが，Latticeのコンパイラは入れない）．

UNIXでは，いくつかデフォルト・ライブラリ（標準ライブラリに加えて）が供給されている（これらのルーチンはUNIXマニュアルのセクション3に書かれている）．この付加されたライブラリは/libか/usr/libディレクトリにある．それらはlibm.a（計算ライブラリ）や，libs.a（標準ライブラリ）の名前を持っている．マニュアルのセクション3にあるサブルーチンを使用するときには，ccかldのコマンドではっきりとライブラリを規定しなければならない．ライブラリの名前は，マニュアルのページの見出しのところにある．例えば，qsort( )の見出しは次のようになっている．

```
QSORT(3C)            UNIX 5.0            QSORT(3C)
```

qsort( )は，libc.a（見出しにはCがある）にあり，ldかccのコマンドに-lcをおくことによってプログラムにlibcがリンクされる．-lXは文字列/lib/libX.aの簡略形である．-lcと/lib/libc.aは同じに扱われる．もしldが/libにそのライブラリをみつけられなければ，/usr/libでも探すだろう．したがって，-lmと/usr/lib/libm.aも等しい．探しているライブラリが/libまたは/usr/libディレクトリになければ，コマンドにフル・パスネームを書き出さなければならない．qsort( )

を使用している sort. c というプログラムの cc コマンドの例は

```
cc sort.c -lc
```

である．-lc は，libc にあるサブルーチンを呼ぶ，すべてのモジュール名の後になければならない．リンカは，それらサブルーチンへの参照をみつけるまで，ライブラリからどのサブルーチンを必要とするのかわからない．それで，リンカはライブラリを走査する前に，ライブラリ関数を参照するすべてのモジュールを処理しておかなければならない．

1.1.5.1　ライブラリ・メンテナンス

ライブラリは，アーカイブ（archive）とも呼ばれる．したがって，UNIX ライブラリ管理プログラムは，ar と呼ばれる．アーカイブは，単一のファイルに編集されたファイルの集合である．それは，C の目的モジュールを含む必要はない．けれども，MS-DOS ライブラリはそれほど汎用性はない．MS-DOS のライブラリは，汎用のアーカイブによってではなく，ライブラリアン（librarian）という，特別なユーティリティでつくられている（DOS では，さまざまなファイル維持プログラムが使用可能であるけれども，それらはライブラリをつくることはできない）．

アーカイブは，ライブラリ以外にも使用される．サブ・ディレクトリを持たない CP/M システムでは，ディスクをきれいにするのを助ける．例えば，ひとつのライブラリの全部のソース・ファイルをひとつのアーカイブに組み込む．その後は，ディレクトリ内のひとつの要素にみえる．また，アーカイブはモデムを通して他のコンピュータにファイルを転送するのにも役立つ．たくさんの小さなファイルを送るより，ひとつのアーカイブ・ファイルを送ることができる．アーカイブ・ファイルを使う他の主な理由は，ディスクの記憶を節約できることである．たいていのオペレーティング・システムでは，たとえ，ファイルがたくさんの記憶を必要としなくても，すべてのファイルが最小の記憶単位を要求する．例えば，最小限の 512 バイトのディスク・セクタが，1 文字分の長さのファイルを保持するのに必要とされる．たいていのオペレーティング・システムがひとつのディスク・セクタを使用していないので，問題はさらにひどくなる．むしろ，最小の記憶割当て単位として，いくつかのセクタからなるクラスタを使う．このように，オペレーティング・システムが 4 セクタのクラスタを使用していれば，4*512,

すなわち，2 048 バイトが1バイトのファイルを収納するために使われる．アーカイブは，いくつかの小さなファイルからつくられるので，浪費する記憶量を最小にする．

前にも述べたが，UNIX のアーカイブ管理ユーティリティは ar という．ar でアーカイブの生成，アーカイブへのファイルの挿入，アーカイブからのファイルの削除，ファイルの出力（アーカイブからファイルを削除しないで），アーカイブ・ファイルの入換え，そして，そのほかの簡単な管理作業ができる．

4つのファイル，one. c，two. c，three. c，four. c があるとする．次のコマンドで，これらのファイルを含むアーカイブ・ファイルをつくることができる．

```
ar rv files.arc one.c two.c three.c four.c
```

第2の引数（rv）は ar にするべきことを教える．ここで，r は指示されたファイルを入れ換えることを意味する．ファイルがアーカイブになければ，アーカイブにそのファイルを加える．通常，UNIX のユーティリティでは引数の前にあるマイナス記号がここでは使われていないことに注意する．v は ar に実行中の内容を知らせるように指示している．コマンドの次の引数 files. arc はアーカイブ・ファイルの名前である．したがって，残りの引数がこのコマンドが使用するファイルである．ファイルはアーカイブにコピーされる．つまり，ファイルは処理の一部として削除されない．

マイクロソフト社のライブラリアンでのコマンド（lib と呼ばれる）は

```
lib files.lib +one.obj +two.obj +three.obj +four.obj ;
```

である．4つのモジュール，すなわち，one. obj，two. obj，three. obj，four. obj を含む files. lib というライブラリをつくる．

さて UNIX に話を戻す．ファイル three. c を次のコマンドで調べることができる．

```
ar p files.arc three.c >there.c
```

ファイルは標準出力に印刷される．それで，アーカイブからファイルを抜き出したいのならば，出力の切換えを行わなければならない．マイクロソフト社のライブラリは，目的モジュールだけを含むことができるから，lib とは同一のコマンドではない．

```
ar tv files.arc
```

は，アーカイブ内のファイルの全情報を出力する．v はロング形式で出力させる．

vがなければ，ファイル名だけが出力される．DOSでの同様のコマンドは

```
lib files.lib,,files.ndx;
```

である．コンマで区切られた空白はlibに行うべき作業がないことを指示している．しかし，3番目のフィールドでlibにファイルfiles.ndxにインデックスをつくるよう指示している．

```
ar d files.arc three.c
```

はアーカイブからthree.cを削除する．マイクロソフト社での，同様のコマンドは

```
lib files.lib -three.obj;
```

である．

いくつか他にも，ライブラリに関連したプログラムが，UNIXでサポートされている．例えば，lorderとtsortはarとともに動作して，順序よく並んだライブラリをつくる．lorderは，目的モジュールの集合を入力とする．モジュール内のサブルーチン間の呼出し関係を決定し，呼出し関係のあるファイル名を対にして出力する．その出力は，tsortに渡される．tsortは，目的ファイル名を並びかえる．そのために，モジュールがその順序でライブラリに挿入されるならば，前方参照はなくなるだろう．

例えば，3つのファイル，larry.o，curly.o，moe.oからなるライブラリをつくりたいとする．これらのファイルのおのおのが，単一のサブルーチン [larry( )，curly( )，moe( ) と呼ばれる] でできている．サブルーチンmoe( )はlarry( )を呼び，larry( )はcurly( )を呼んでいる．したがって，moe( )はlarry( )やcurly( )より前にライブラリに挿入されていなければならない．同様に，larry( )はcurly( )より前に挿入されていなければならない．このコマンド

```
lorder larry.o curly.o moe.o | tsort
```

は，次の表を生成する．

```
moe.o
larry.o
curly.o
```

この出力には，moe( )に続いてmoe( )に呼ばれるサブルーチンを含んだモジュールlarry( )がある．同様に，続いてcurly( )がある．lorder/tsortからの出力を次のコマンドでarに渡すことができる．

```
ar cr stooges 'lorder larry.o curly.o moe.o | tsort'
```

ここではバック・クオートを使っており，通常のシングル・クオートではない

ことに注意するべきである．

ライブラリが順序よく並んでいることの利点は，リンカに，よりはやく走査されるので，コンパイル時間を短縮できることである．けれども，いつも順序よく並んだライブラリを持つことができるとは限らない．そのために，UNIX には **ranlib** 機能が与えられている．ranlib は，でたらめに並んだライブラリの名前を入力とする．ranlib は，リンカが必要とする関数を探すために使うシンボル・テーブル（--.SYMDEF と呼ばれる）をライブラリに挿入する．例えば

```
ar cr stooges larry.o curly.o moe.o
ranlib stooges
```

は，でたらめに並んだライブラリ stooges をつくり，リンカがすべての必要な関数をアクセスすることができるように，そのライブラリを修正する．

マイクロソフト社のライブラリはでたらめに並んでいてもよい．けれども，順序よく並んだライブラリのほうがでたらめなものよりもはやくリンクできる．だから，順序よくライブラリをつくることは価値がある．残念ながら，私は MS-DOS のもとで走る lorder や tsort については知らない．

1.2　モジュラ・プログラミング

モジュールは，C プログラムの重要な部分である．だから，どのようにプログラムのモジュール構成をするか考えることが重要である．ライブラリ・モジュールはできるだけ小さくつくるべきである．これは，大きなモジュールでプログラムをつくるのが大変だからではない．小さいモジュール構成の方が，維持管理の面でよいからである．ひとつのモジュールにあるサブルーチンは，すべて機能的に関連しているべきである（デバッグ中に，ひとつのファイルから別のファイルへと移る必要がないようにするべきである）．十分に汎用性のあるサブルーチンは，大きなファイルに埋め込むよりも，ライブラリにおくべきである．

モジュールをいっしょにするときに重要なことは，グローバル変数とマクロの配分である．できれば，グローバル変数はひとつのモジュールに通用範囲を限定するべきである．つまり，グローバル変数は他のモジュールにあるサブルーチンに使われるべきではない（これは，維持管理のために第 1 にされることである．5 章で詳しく討議する）．同じことが **#define**，**typedef** などにもいえる．つまり，できればそれらもひとつのモジュール内でのみ使われるべきである．しかし，時

にはこのような構成はできない．それで，C は **#include** 命令を用意している．#include <files> 文は，ソースコード内に指定されたファイルの内容全部と置き換えられる．そのように使われるファイルはインクルード・ファイルと呼ばれ，そのファイル名は .h で終わらなければならない．

インクルード・ファイルには，決して実行可能コードや変数宣言を含むべきではない．それらは，.c ファイルにおいてコンパイルし，通常のやり方で最終プログラムにリンクするべきである．インクルード・ファイルには **#define**，**typedef**，**extern** 文だけが含まれるべきである．**#include** 文は入れ子構造（nesting；.h ファイルにまた **#include** 文をおく）ことができるけれども，維持管理の理由から一般によい考えではない．少なくとも2つ以上のファイルに使われるのでなければ，インクルード・ファイルに書くべきではない．逆に，2つ以上のファイルに使われるすべての #define，typedef，extern 文は，全ソース・ファイルに組み込まれる .h ファイルにおかれるべきである．グローバル変数などは，プログラム中に散在させるよりも，ひとつの場所に集中させるほうが変更しやすい．

1.3　他の役立つプログラム

C プログラマの仕事を，より簡単にする UNIX ユーティリティをいくつか取り上げる．これらには MS-DOS 版もある．

1.3.1　grep

grep は，文書ファイル中のパターンを探すプログラムである．パターンは，特に強力な型で規定されている．その型は“ページのもっとも左のカラムから始めて，空白やタブ，開かっこ，いろいろな文字の繰返し，閉かっこ，セミコロン以外のものが続く a から z の範囲の文字列”の発見を可能にする．grep は，複数ファイル構成の C プログラムでファイル間参照をつくるのに使うことができる．リンカが未解決の参照だと思うようなつづりの間違ったサブルーチン名をみつけたり，製作中のオペレーティング・システムを構成する 45 個のファイルのうちのひとつにサブルーチンの宣言がないことをみつけたりするのにも役立つ．

grep は正規表現（regular expression）と呼ばれる表記法を使用してファイルの内容と照合してパターンを探す．正規表現は照合する文字の組合せとメタキャラクタ（metacharacter）と呼ばれる特殊記号からできている．メタキャラクタを次

にあげておく．

^ 行の始まりと照合する．

$ 行の終わりと照合する．

/ 次にある文字と照合する．この場合，/* はアスタリスクを照合し，/. はピリオドを照合する．

. どんな文字とも照合する．

[ ] [ ] で囲まれた文字列が，文字の種類（character class）を規定する．その文字列内のどの 1 文字とでも照合する．例えば，[abc] は a，b，または c と照合する．ASCII 文字コードの範囲は [a-z0-9] のように省略形で書くことができる．[に続く最初の文字が ^ ならば，範囲外の文字（negative character class）を規定する．この場合，[ ] で囲まれた文字範囲外のすべての文字と照合する（いいかえれば，[^a-z] は英小文字以外のすべての文字と照合する）．範囲外の指定は，必ず何かと照合しなければならない．しかし，その何かは範囲内の文字ではない．例えば，^$は ^[z] $ と同じではない．最初の例は空行（行の始まりに行の終わりが続いている）を照合する．2 番目の例は z 以外の文字があって，行の終わりが続く行の始めと照合する．2 番目の例では，行に文字がひとつなければならない．しかし，その文字は z では駄目である．*．^ と $は [ ] の中では特殊文字ではないことに注意すること．

* * の前の正規表現と同じものがあってもなくてもよい．

＋ ＋の前の正規表現と同じものが 1 回以上ある．

ee 連結された 2 つの正規表現の，最初の正規表現に 2 番目の正規表現が続いているところがあるか照合する．

| | か改行で分けられた 2 つの正規表現の最初か 2 番目の正規表現と同じものがあるか照合する．

優先順位は [ ]，*，連結，|，改行である．greedy アルゴリズムが * と ＋ を処理するために使われている．それで，a.*z は左端が a で始まり右端が z で終わる文字列を照合する．その間には，a や z を含めて文字がいくつかあってもよい．たいていの UNIX には，grep，egrep，fgrep と呼ばれる grep の 3 つの変形版がある．それらは，コマンドがわずかに異なるので，詳しい使用方法は自分のマニュアルを引くべきである．grep 自身は，限定された正規表現文しか認識しない（例

えば，メタキャラクタ | を認識しない）．egrep はさきにあげたすべてのメタキャラクタを認識する．概して，grep が3つの中でもっとも便利で，egrep がもっとも強力である．

いくつか例をみてみよう．コマンド

```
egrep -nf index.exp *.c
```

は，大きなCプログラムのファイル間の参照表をつくる．

ここで，ファイル indef. exp の内容は

```
^[a-zA-Z_]+.*([^;]*)[^;]*$
^#define[    ]+[a-zA-Z_]+(
```

である．ファイル間の参照表は，サブルーチン名とサブルーチンのようなマクロの名前を含む．名前が .c で終わるすべてのファイルが調べられる．各出力行には，その行があったファイルの名前（2つ以上のファイルを調べたときは自動的に行われる）と行番号（-n があるので）がさきに書かれる．

indef. exp には2つの行があるから，どちらの正規表現にもマッチするすべての行が出力される．正規表現は，以下のように解釈される．

```
^[a-zA-Z_]+.*([^;]*)[^;]*$
```

は，行の始めに（ ^ ），文字か下線（[a-zA-z_]）が一回以上（＋）あり，またどんな文字でもゼロ回以上あり（.＊），開カッコが続き（（ ），セミコロン以外の文字をゼロ回以上繰り返して（[^;]＊），閉カッコが続き（)），セミコロン以外の文字をゼロ回以上繰り返して（[^;]＊），行が終わる（＄）．

```
^#define[    ]+[a-zA-Z_]+(
```

は，行の始まりに，文字列 ＃define があり（＃define），少なくともひとつの空白かタブが続き（[ ]＋：[ ] の間に空白かタブ記号（^I）がある），少なくともひとつは文字か下線が続き（[a-zA-Z_]＋），すぐに（区切りの空白をおかずに）開カッコが続く．

grep は通常，コマンドに正規表現を書いて呼び出される．シェルにとっても特別な意味を持つ ＊ や | の使用には注意すること．どちらの文字も前にバックスラッシュをおくか，または，全体の表現をクオートで囲まなければならない．例えば

```
grep "extern.*foo" *.c
```

は，foo の全 extern 宣言を求めてすべてのファイルを探す．式中の ＊ は，クオートがあるためにシェルには影響しないだろう．しかし，＊.c の ＊ はシェルに解釈

される．さきの例は，次のコマンド行でも実行できる．

```
egrep "^[a-zA-Z_]+.*([^;]*)[^;]*$|^#define[    ]+[a-zA-Z_]+(." *.c
```

ここで，|は元のファイルにあった改行と同じ機能を持つ．

1.3.2 lint

lint は，潜在的な問題を突き止めるのに，通常のコンパイラよりももっと厳密にCプログラムのチェックを行う．lint は，サブルーチンに渡す引数の個数の間違い（あるいは誤った型の引数）や，または，正しい型に割り当てられないサブルーチンの返り値のようなものをみつける．lint は演算子の通常でない使用（条件文で == の代わりに = を使う）や，初期値の代入されていない変数の参照，暗黙の型変換によって起こる切捨て等を潜在的なエラーとして示す．lint は，単純なエラーをみつけるのに役立つ．一方，完全に問題のないプログラムにエラー・メッセージや警告を発生することもある．そのため，アプリケーションによってはまったく役立たない．

lint は，実際にはコンパイラの変形版である．lint は，出力としてコードよりもエラー・メッセージを生成する．そのため，コンパイラが使用される場合とまったく同じように使われる．ひとつの違いは，lint はソース・コード上で操作しているので，リンカと同じではないことである．プログラム中のすべてのモジュールを同時に lint に通したくなければ，本当のファイルの代わりに仮のサブルーチン宣言を含んだ特別なファイルを lint に与えることができる．その仮のサブルーチン宣言は次のようなものである．

```
long  foo(a,b) int a; long b; { return 0L; }
```

この仮の宣言は，エラー・チェックをするために必要なことをすべて lint に知らせている．これは，完全なサブルーチンよりもはやく処理される．

1.3.3 Make

make は実際に使用するまでは，どうしてわざわざそれを使うのかわからないという，ユーティリティのひとつである．そして，使ってからは make なしではどうしたらよいかわからなくなる．make は，プログラムの自動コンパイレイション・ユーティリティである．make は 47 モジュールで構成されているある巨大なプログラムの中で再コンパイルしなければならないものを（与えられたルールか

ら）決定し，それらのモジュールだけを再コンパイルする．また，インクルード・ファイルを書き換えたとき，再コンパイルしなければならないものを区別することもできる．make は，依存関係を保持することによって，これらすべてを行う．C のソース・ファイルよりも，後に修正されたインクルード・ファイルがあれば（または，C のソース・ファイルが目的ファイルより後で修正されていれば），そのファイルは再コンパイルされなければならないことを make に教えておく．make は，複数のファイルに分割された大きなプログラムを維持管理するために，貴重な手助けになる．幸いなことに，make には MS-DOS 版もいくつかある．そのため，make を使うために UNIX は必ずしも必要ではない．

make は makefile を入力とする．makefile は大きなプログラムのすべてのモジュールとそのモジュール間の関係を記述しているファイルである．makefile を使用して，make はどのモジュールを再コンパイルするべきか決定し，そして，それらのモジュールだけを再コンパイルする．make は，C プログラムと C コンパイラがどのように動作するのか知っていて，絶対必要な作業だけを行うインテリジェント・バッチ・ユーティリティのようなものである．

make を説明するもっともよい方法は，例をあげることである．farm という目的プログラムをつくりたいとしよう．もとのソース・プログラムは，3つのファイル，cow. c，pig. c，farm. c に分割されている．3つのファイル全部が，＃include <stdio. h> 文を含んでいる．さらに，cow. c，pig. c は ＃include “animals. h” 文を含んでいる．さて，cow. c にある何かを変更したとしたら，cow. c を再コンパイルして，新しい cow. obj を farm. exe に再リンクしなければならないだろう．

```
#
# Make farm using cc
#
farm:   cow.o pig.o farm.o
        cc -o farm cow.o pig.o farm.o

cow.o:  cow.c  stdio.h animals.h
        cc -c cow.c

pig.o:  pig.c  stdio.h animals.h
        cc -c pig.c

farm.o: farm.c stdio.h
        cc -c farm.c
```

図 1・4　簡単な makefile

animals.h の何かを変更するとしたら，cow.c と pig.c を再コンパイルして，再リンクしなければならないだろう．stdio.h を変更したとすれば，すべてを再コンパイルする必要があるだろう．make は，依存関係として，これらのファイルの関係を記述する．makefile は，この依存関係の表である．今述べたプログラムの makefile を図 1･4 に示す．

＃はコメント行を示す．farm.exe は cow.o，pig.o，farm.o からできている．だから，cow.o，pig.o，farm.o のうちいずれかが farm より後で変更されたら，farm は再生成されなければならない．この再生成は，cc -o farm cow.o pig.o farm.o が実行されることによって行われる．同様に，ファイル pig.c，stdio.h，animals.h のいずれかが pig.o より後で変更されれば，pig.o は，コマンド cc -c pig.c で再生成される必要がある．-c は，目的モジュールをつくってリンクはしない．おのおのの依存関係に関連した作業は数行に及ぶ．この処理を行わせるには，make コマンドをタイプするだけでよい．make は makefile を読み，そこにある情報を使って何をするべきか理解して，それを行う．

マイクロソフト社のコンパイラや，MS-DOS のコンパイラを使用しているならば，たったいま引用した例では farm は farm.exe と呼ばれる．同様に，すべての .o ファイルは .obj という名前になる．最後に，プログラム名 cc は，マイクロソフト社のドライバ・プログラム名 cl に変更される．それ以外は，makefile は同様である．

make は，作業を少し簡単にするオプションをいくつかサポートしている．マクロは，タイプする量を減らすために使われるかもしれない．マクロは

```
INCLUDES = stdio.h animals.h
OBJECTS  = cow.o pig.o farm.o
COMPILE  = cc -c

farm:    $(OBJECTS)
         cc -o farm $(OBJECTS)

cow.o:   $(INCLUDES) cow.c
         $(COMPILE)  cow

pig.o:   $(INCLUDES) pig.c
         $(COMPILE)  pig

farm.o:  stdio.h     farm.c
         $(COMPILE)  farm
```

図 1･5 マクロを使用した makefile

```
name = stuff
```

で定義される．ここで，name はマクロ名であり，stuff は置き換えられるテキストである．このマクロは，$(name) で置き換えられる．マクロを使用した makefile を図1･5に示す．

この makefile は，もっと簡単にできる．すべての .c ファイルが，同じ手順で .o ファイルに変換されることに注意すること．make は，このような同じ手順を繰り返し行うために，依存関係の一般化（generic dependency）と呼ばれる機構がある．図1･6に示す makefile を考える．

```
#
# Make farm using the Lattice C compiler.
#

INCLUDES = stdio.h animals.h
OBJECTS  = cow.o pig.o farm.o

.c.o:
        cc -c $*.c

farm:   $(OBJECTS)
        cc -o farm cow.o pig.o farm.o

cow.o:   $(INCLUDES)
pig.o:   $(INCLUDES)
farm.o:  stdio.h
```

図 1･6　汎用の makefile

```
.c.o:
        cc -c $*.c
```

の行は，次のことを意味している．.c ファイルから .o ファイルをつくるために，

```
cc -c $*.c
```

を実行する．$＊ は，先に定義したマクロである．それは，つくられるファイル名（関係する行のコロンの左側にある）のルート部分（イクステンションを除いた部分）になる．すなわち，次のような関係する行，

```
foo.o: foo.c rat.h
```

があるとすると，$* は文字列 foo になる．この依存関係の一般化の場合は，make はすべての .o ファイルが同じルート名を持つ .c ファイルに依存していると想定する．すなわち，foo.obj ファイルがつくられるときには，foo.c との依存関係が想定され，関係する行に foo.c がある必要はない．

コンパイレイション処理を始めるには，ただ，make とタイプすればよい．依存関係は，make によって調べられ，そして，適切な動作が行われる．make は，コマンド上に処理したいファイル名を指定すると，その目的ファイルだけを再コンパイルすることもできる．例えば

```
make pig.o
```

は pig. c だけを再コンパイルする（pig. o をつくり直す必要があれば）．

2 つのオプションが特に有用である．オプション -t（touch(訳注 2) の意味）は，まるでプログラムを再コンパイルしたかのように最後の更新時刻をかえる．実際にはコンパイレイションは行われていない．このオプションはコメントだけを変更して，プログラム全体を再コンパイルしたくないときなどに有効である．オプション -i は，make に呼び出したコマンドから返されたエラー・コードを無視させる．通常，コンパイラが make にエラーを返すと，コマンドは終了する．もしコマンド上に -i が規定されていれば，make は実行を続ける．このオプションは，長い時間がかかるコンパイルを始めて，昼食にでも行きたいときに役立つ．コンパイルは，たとえエラーがあっても続けられる．必ず make の出力をファイルに切り換えておくこと（いいかえれば，make -i >& err）．さもなければ，エラー・メッセージはスクリーン上を消えてゆき，2 度とみることができなくなるだろう．

1.4　練　　　習

1-1　getchar( ) を使用して 1 行のテキストを読んで，putchar( ) を使用してその行をプリントするプログラムを書きなさい．プログラムは 3 つのファイルに分ける．入力ルーチンを含むファイルと，main( ) を含むファイルと，出力ルーチンを含むファイルである．ファイルは，別々にコンパイルして，いっしょにリンクしなさい．

1-2　読者のコンパイラとともに供給されているライブラリアンを使って，練習問題 1-1 で書いた，入力と出力のルーチンを含むライブラリをつくりなさい．main( ) を含むモジュールを，ライブラリとリンクして完全なプログラムにしなさい．

1-3　練習問題 1-1，1-2 のすべてを行う makefile を書きなさい．それから，make

訳注 2：指定したファイルのファイル更新日時を現在の日時に変えるコマンド

を使って実行しなさい.

1-4 読者のソース・ファイルを lint に通しなさい. どんな出力が生成されるか? ソース・ファイルに 2, 3 のエラー(使用していない変数を宣言したり, 型と個数を間違ったサブルーチンを呼ぶ)を加えて, 再び lint に通しなさい. lint は何をするか?

1-5 C プログラム中に非静的なグローバル変数の宣言を探す grep のコマンドを書きなさい. つまり, grep は

```
int     Jack ;
long    Jill ;
char    *hill[20];
```

を探す. しかし

```
int     Alice ( )       /* no ; */
int     Humpty ( ) ;
static  int  Tweedledum;
extern  long Tweedledee( );
```

は探さない.

2　2進数演算

この章では，初歩的な2進数演算と，数を計算機内で表すやり方によって起こる，読者が出合う問題を探る．途中，Cの演算子について述べる．

2.1　基数と指数

たいていの場合，2進数システムでの演算は，10進数システムでの演算と同じように行われる．このことをしっかりと覚えておけば，多くの潜在的な混乱を避けられるだろう．どんな基数の数でも，その数内の位置が，実際の値を決定する各桁の和からできていることを，小学校3年生で覚えたかもしれない（本章では，小学校3年生の計算について多く述べるつもりはない）．さらに，どの桁の数字も，その数の基数と同じでも大きくもない．例えば，8進数（基数は8）のすべての桁の数字は0から7の範囲である．2進数（基数は2）は，0または1からできている．基数が，16の数は9より大きな値の桁のために特別な記号を必要とする（A—Fが使われる）．

数全体の値は，桁の位置によって決まる指数の回数だけ，基数を乗じた各桁の値の和である．10進数418は実際には

```
400 + 10 + 8
```

の和である．別の表現では

$$(4 * 10^2) + (1 * 10^1) + (8 * 10^0)$$

の和である（C言語では，＊は乗算を意味する）．2進数（基数は2）は，各桁を計算するのに，10の指数ではなく2の指数が使われること以外は，10進数と同様である．例えば，2進数1101は

$$(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) =$$
$$(1 * 8) + (1 * 4) + (0 * 2) + (1 * 1) =$$
$$8 + 4 + 0 + 1 = 13$$

の値である．

10進数（基数10）	2進数（基数2）
数：1985	数：1010
1 * 1000 = 1 * 10^3	1 * 8 = 1 * 2^3
9 * 100 = 9 * 10^2	0 * 4 = 0 * 2^2
8 * 10 = 8 * 10^1	1 * 2 = 1 * 2^1
5 * 1 = 5 * 10^0	0 * 1 = 0 * 2^0

図 2・1　10進数と2進数

2^0 = 1	2^7 = 128	2^{14} = 16,384
2^1 = 2	2^8 = 256	2^{15} = 32,768
2^2 = 4	2^9 = 512	2^{16} = 65,536
2^3 = 8	2^{10} = 1,024	2^{17} = 131,072
2^4 = 16	2^{11} = 2,048	2^{18} = 262,144
2^5 = 32	2^{12} = 4,096	2^{19} = 524,288
2^6 = 64	2^{13} = 8,192	2^{20} = 1,048,576

図 2・2　2の累乗

10進：	2進：	16進：	8進：	10進：	2進：	16進：	8進：
0	00000	0x0	000	16	10000	0x10	020
1	00001	0x1	001	17	10001	0x11	021
2	00010	0x2	002	18	10010	0x12	022
3	00011	0x3	003	19	10011	0x13	023
4	00100	0x4	004	20	10100	0x14	024
5	00101	0x5	005	21	10101	0x15	025
6	00110	0x6	006	22	10110	0x16	026
7	00111	0x7	007	23	10111	0x17	027
8	01000	0x8	010	24	11000	0x18	030
9	01001	0x9	011	25	11001	0x19	031
10	01010	0xa	012	26	11010	0x1a	032
11	01011	0xb	013	27	11011	0x1b	033
12	01100	0xc	014	28	11100	0x1c	034
13	01101	0xd	015	29	11101	0x1d	035
14	01110	0xe	016	30	11110	0x1e	036
15	01111	0xf	017	31	11111	0x1f	037

図 2・3　数

2進と10進の間の違いは，各桁にかけられる乗数だけである．この処理は，図2·1に書かれている．2の指数は，図2·2に書かれている．文字K（kiloの意味）は，2^{10}を表すのに使われている（10Kバイト＝10＊1024＝10240バイト）．

厳密にいえば，16進数（しばしばhexと呼ばれる）と，8進数（octal）は，それぞれ基数が16と8の数であるけれども，実際は2進数演算のことについて述べるときに簡略形として使われている．C言語では，まえに0xがついているすべての数は16進数である．8進数は0（xはない）を前に付けている．図2·3は10進，2進，8進，16進数間の相対表である．

2.2 基　数　変　換

16進数と8進数は，異なるやり方で2進数を分割して簡単につくられる．8進数は，3桁の2進数からつくられる．例えば

```
10110011 = 10,110,011  = 0263 （8進数）
```

2進数　10は8進数で2
2進数110は8進数で6
2進数011は8進数で3

である．この処理は，4桁のグループが使われること以外，16進数でも同じである．すなわち

```
10110011 = 1011,0011  = 0xb3 （16進数）
```

2進数1011は16進数でb
2進数0011は16進数で3

である．

2進数から10進数への変換は，各桁に2をその桁の指数回かけ，それぞれの桁の結果を合計してできる．16進数から10進数への変換は，その桁の指数回16をかけること以外同じである．この処理を図2·4に表す．

```
------ 1 * 8  =   8        ------ (a * 16³) = (10 * 4096)  = 40960
|----- 0 * 4  =   0        |----- (6 * 16²) = ( 6 *  256)  =  1536
||---- 1 * 2  =   2        ||---- (f * 16¹) = (15 *   16)  =   240
|||--- 0 * 1  = + 0        |||--- (2 * 16⁰) = ( 2 *    1)  =     2
||||            -----      ||||                              -------
||||               10      ||||                                42738
1010                       a6f2
```

図 2・4　2進数/16進数から10進数への変換

10進数から2進数への変換はもっと難しい．変換には2つの方法がある．どちらも有効ではあるが，適用が異なる．method1は，変換された数をスクリーンに書いたり，配列に入れたりするルーチンにはよい．method2はそのほかに適用するのによい．どちらのアルゴリズムでも，2進数の各桁は次のように表す．

D15, D14, ... , D1, D0.

D15は最上位ビットであり，D0は最下位ビットである．

```
Method 1:

    (1) D15 = Number / 2^15
    (2) D14 = (1)の除算の余り / 2^14
    (3) D13 = (2)の除算の余り / 2^13
    (4) D0まで繰り返す.

Method 2:

    (1) numberが奇数ならばD0=1, 偶数ならばD0=0
    (2) number = number/2 ;
    (3) numberが奇数ならばD1=1, 偶数ならばD1=0
    (4) number = number/2 ;
    (5) D15まで繰り返す.
```

このアルゴリズムで，numberは変換すべき数，/は整数除算である．商の小数部分は切り捨てる．つまり，1/4＝0（1/4＝0.25であるが，小数0.25は切り捨てる）である．5/2＝2（5/2＝2.5であるが，0.5は切り捨てる）である．余りは，除数の点から表される商の分数部分である．つまり，7/5＝1＋2/5であれば，1は整数の商で，余りは2（2/5より）である．別の例では，17/6＝2＋5/6（2が商で，5が余り）である．剰余演算子（しばしばMODと略される）は整数わり算の余りを生ずる．Cでは，%が剰余演算子として使用される．すなわち，17/6＝2，17%6＝5である．

2.3　加 算 と 減 算

前に述べたように，2進数演算は10進数演算と同様に行われる．10進数は次のように加算される．

（0）　右側の欄を縦にたす．

（1）　結果が9より大きかったら，次の欄へ繰り上げる．

（2）　次の欄を，繰り上がりの数（キャリー）を含めてたす．

（3）　もっとも左の欄までこの方法を続ける．

2進数も同様に行われる．

0+0は0でキャリーは0，0+1は1でキャリーは0，1+1は0でキャリーは1，そして，1+1+1は1でキャリーは1である．2つのもっと実際的な加算の例を図2·5に示す．減算をするには，ひとつの数を負の数にして加えなさい．

```
キャリー：  1   1            キャリー：  111111
         00101001 = 41              00101111 = 47
        +00100101 = 37             +00011101 = 29
        ----------  ----            ---------  ----
         01001110   78              01001100   76
```

図2·5　加　算

C言語で加算が行われるとき，最上位ビットからのキャリーは捨てられる．これは問題を起こすこともある．例えば，8ビットの符号なしの数255は**char**の大きさの変数(0xffの値を持つ)に格納される．しかし

```
unsigned char   x ;

x = 255;
z = x + 1 ;
```

は加算の結果0になる．次の2進数演算をみれば理由がわかるだろう．

```
キャリー： 1 1111 111
             1111 1111 = 255
         +   0000 0001 =   1
         ------------    ---
           1 0000 0000     0
```

もっとも左のビットは，8ビットに収まらないので，捨てられる．

2.4　負　　数

負数の概念は，10進数システムでだけ意味を持っている．負の10進数を表すために2進数，16進数，8進数を使うことができる．しかし，2進数自身に負の数はない．すなわち，0x1a2は10進数で418であるけれども，−0x1a2は意味がない．0xfe5eは16ビットの計算機で10進数の −418を表している．負の数は，

たいていの計算機では，2の補数として知られる方法で表されている．負の2の補数をつくるには，正の数のすべてのビットを反転させ（1は0に，0は1にかえる），その結果に1を加える．その過程を図2·6に示している．

```
              1 = 00000001         27 = 00011011         88 = 01011000

反転：            11111110              11100100              10100111
加算1：         + 00000001            + 00000001            + 00000001
                ----------            -----------           ----------
             -1 = 11111111        -27 = 11100101        -88 = 10101000
```

図 2·6　負数の形成

2の補数には，いくつか面白い特質がある．負数を打ち消して正数にかえることができる．もっと重要なことは，さまざまな算術演算がすべてできることである（つまり，負数と正数をたしたり，ひいたりして正しい符号の結果を得ることができる．正数に負数をかけて負数を得る，などができる）．

負数のもっとも左のビットは，常に1であることも重要である（正数では常に0である）．このために，2進数のいちばん上のビットは，符号ビット（sign bit）と呼ばれる．符号ビットのために，ハグがいくつか現れることもある．次は8ビット計算機での例である．

```
キャリー： 11
           00101000 =   40
         + 01100100 =  100
         ----------    ----
           10001100 = -116
           |
           └── 上位ビットが1，それで
               数は負数である．
```

これは期待していたものではない．8ビットではなく，16ビットを使用していたならば，このエラーは起こらなかっただろう．計算は次のようになる．

```
キャリー：           11
           0000000000101000 =   40
         + 0000000001100100 =  100
         ------------------    ----
           0000000010001100 =  140
```

負数になる可能性があるときには，2進数は，計算機の十分な語長に引き伸ばして表されなければならない．

負数を表す方法が及ぼした別の重要な作用は，符号拡張（sign extentsion）である．数がより長い型に変換される（例えば，8から16ビットへ）ときには必ず元

の数の符号は維持されなければならない．これは，新しく加えられた上位ビットに，短い数の方の符号ビットを2重にして行う．8ビットと，16ビットで −27 を考えてみよう．

```
-27 =         11100101  ( 8 ビット)
-27 = 1111111111100101  (16 ビット)
```

8ビットの数が拡張されるとき，短い方の最左のビットが，長い方の最左バイトの全ビットに2重化されている．下位バイトの符号ビットが上位バイトに拡張されている．

符号拡張は，C プログラムに発見するのが難しいバグを生みだす．これは10章と本章の後半でも述べる．

2.5 シフト，乗算，除算

2.5.1 シ フ ト

10進数の計算では，左にシフトして10をかけることができる．すべての桁がひとつ左へ移動し，もっとも右の桁には0が入れられる（60はひとつ左にシフトして600になる）．同様に，10進数はひとつ右にシフトして，10でわることができる（60はひとつ右にシフトして6になる）．この処理はどの基数にも適用できる．すなわち，基数をかけるにはひとつ左に桁をシフトし，基数でわるには右にシフトする．計算機は2進であるから，左シフトは2の乗算，そして，右シフトは2の除算である．C では，演算子≪と≫はそれぞれ左，右シフトを表すのに使われる（これらをより大きい（>）や，より小さい（<）と混同しないように注意すべきである）．N≪2は，2進数の2桁分左にシフトした（4をかけた）N と等しく，N≫X は，X ビット右へシフトした（2^X で割った）N と等しい．ここで，≪や≫は演算数を更新しないことに注意しなさい（＋，− やその他の演算子がする以上に）．上の例で実際に N を更新するためには，記号 ＝ が必要である（言いかえれば，N≪＝2，または，N＝N≪2 とする必要がある）．数が左にシフトされるときには，最下位ビットには0が入れられる．右シフトでは，上位ビットは通常2重化される（シフトされた数の符号を維持するため）．例えば，8ビットの2進数10001101を，左に1ビットシフトすると00011010になる．10001101を1ビット右にシフトすると11000110になる．理論上は，右シフトの符号拡

張は認められない．しかし，たいてい，どのコンパイラでもそれを行っている．

2.5.2 乗　算

経験では，計算機はかなり馬鹿であると言える．たいていの計算機は，根本的に加算機を実際以上によくみせている．計算機ができることのすべては簡単な計算とメモリのやりとりである．いくつかの計算機（8080/Z-80 ファミリはこの中に入る）は，乗算ができない．しかし，古い型の加算機のようなものは，加算とシフトができる．2つの数は，一方の数字自身を何回か加算することで，かけ算される．実際には，X+X+X を使う方が X*3 よりもはやくできることがある．コンパイラの中には，演算数が十分小さいときに，乗算を行うのに加算を繰り返し（または，加算とシフトを組み合わせて）生成するものがある．しかし，いったん乗数がワード幅よりも大きくなると，連続した加算はかなり遅くなる．やりたいことが2をかけたいだけならば，数をシフトするだけでよいだろう（乗算や除算をするよりも，使えるのならばシフトをするほうが通常はずっと効率がよい）．2以外をかけるには，もう少し複雑になる．2つの数の乗算には，たぶん，次のようなアルゴリズムを使っている．

（0）かけられる数は被乗数である．被乗数にかける数は乗数である．かけ算の結果は積である（式 2*3=6 では，2が被乗数であり，3が乗数で，6が積である）．

（1）被乗数に，乗数の最右の桁をかける．すると部分積ができる．

（2）被乗数に，乗数の次の桁（前のステップの桁の左となりの桁）をかける．かけ算の結果を，左へ1桁シフトして，ステップ1の部分積に加える．

（3）乗数全体が終わるまでこの方法を続ける．

この処理は，次のようにもっときちんと書くことができる．

（0）最下位の桁を0にする．最上位の桁を1にする．

（1）N=0，積=0．

（2）乗数が0に等しいならば，終わり．

（3）被乗数に，乗数のもっとも右の桁をかける．このかけ算の結果をN桁左へシフトして，その結果を積にたす．

（4）乗数を1桁右へシフトし，もっとも右の桁を捨てる．

（5）N=N+1

（6） ステップ2へ行く．

このアルゴリズムは，2進数のかけ算でも使用できる．ただ，桁をビットにかえればよい．このアルゴリズムの問題点は，再帰的に定義されることである．アルゴリズムは，かけ算の仕方を教えているけれども，いつも1か0だけをかけている．この程度の低いかけ算は，まったく本当のかけ算ではない．むしろ，数をある位置から別の位置へ動かすかどうか，あるいは数よりも0を動かすかどうか判断している．2進数のかけ算の過程を図2·7に表している．

```
     00110 =  6
×    01010 = 10
-----------
     00000          (0110 * 0 が左に0ビットシフトされた)
    00110           (0110 * 1 が左に1ビットシフトされた)
   00000            (0110 * 0 が左に2ビットシフトされた)
  00110             (0110 * 1 が左に3ビットシフトされた)
-----------
  00111100 = 60     (4つの部分積の和) l 4 partial prod
```

図 2·7 2進数かけ算

好きずきだが，部分積を全部を1度にたすのではなく，2つずつたすこともできる．

```
     00000     (第1の部分積)
   + 00110     (第2の部分積)
   ---------
    001100     (第1＋第2)
  + 00000      (第3の部分積)
   ---------
   0001100     (第1＋第2＋第3)
 + 00110       (第4の部分積)
   ---------
  00111100     (第1＋第2＋第3＋第4)
```

符号拡張は，2進数の乗算でも問題である．部分積が負なら，その積を十分なワード幅に（8ビットどうしをかけているときは16ビットに）符号拡張しなければならない．この状態を図2·8に表している．

ここで，結果は8ビットに切り詰めなければならないことに注意すべきである．切り詰めなかったら，結果は0111101110＝494になってしまう．また，この方法では計算のために負の演算数が上になければならない（さもないと符号拡張は行われないだろう）．乗算のアルゴリズムには，被乗数と乗数をどちらも正にしてかけ，後から積の符号を合わせることで符号拡張の問題を回避しているものもある．他にも，もっと優れたよい方法があるが，それはこの本の範囲を越えている．

2つの4ビットの数のかけ算，ひとつは負数，8ビットの結果になる．
部分積の最右の4ビット以外はすべて符号拡張．

```
             1010  =  -6
             0011  =   3
        -----------
        111111010
         11111010
       0 0000000
      00 000000
      -------------
         11101110   = -18
```

図 2·8　異なる符号を持った数のかけ算

2.5.3　除　　算

2進数の除算も,10進数の計算を手本にしている.それで,まず10進数での処理をみることは価値がある．次の問題を考えてみなさい．

```
                    __014_      <-- 商
除数  -->  11  )  156        <-  被除数
                     11
                   -----
                      46
                      44
                   -----
                        2       <-  余り
```

長いわり算の処理は，次のように記述することができる．

（0）　式A/B=Cでは，Aは被除数，Bは除数，Cは商である．

（1）　1（156の最左の桁）を11（除数）でわってみる．わることはできない．それで，商に0をおく．

（2）　15（156の左から2桁）を11でわってみる．1回われる．それで，商に1をおき，15から11をひき，4がでる．被除数から6をおろしてきて，46にする．

（3）　46を11でわってみる．4回われる．それで，商に4をおいて46から44（4＊11）をひき，2がでる．

（4）　もうおろしてくる桁がない．それで，処理は終わり．余りは2である．

このアルゴリズムには，いくつか問題がある．大事なことは，作業自体がとても複雑であることに気付くことである．普通の状態では，先頭に0があることは考えない．2進数では考えなければならない被除数が，除数に対応して，いつも

おき直されなければならないことに注意すべきである．さらに，被除数自身が，アルゴリズムの進行につれてかわっている（ひき算をするとき，実際には新しい被除数をつくりだしている）．最後に，全体の被除数をみていないことに注意すべきである．つまり，被除数の小さな部分をみているだけである．

ここで，2進数の例をみてみよう．次はよく慣れた形でわり算を表している．

```
          -----101-
110 ) 100010        (34 / 6 = 5)
        110
       -----
         101
         000
        -----
         1010
          110
         -----
          100       余り = 4
```

余りからはいつも0か除数をひいていることに注意すべきである．また，4桁の被除数以外は，みていないことにも注意しなさい．

除数を右にシフトするよりも，被除数を左にシフトするほうが簡単であることがわかる．被除数全体を操作していない，むしろ，被除数を4ビットのウィンドウにして操作している．アルゴリズムは，進行につれてそのウィンドウを通して被除数をシフトしている．ウィンドウから（左に）押しだされた桁は捨てられる．ウィンドウは0ビットに初期化され，被除数の上位3ビットが続く．

このアルゴリズムは，2Nビットの被除数をNビットの除数でわって，Nビット幅の商をつくる．ウィンドウは，被除数の上位4ビットからつくられたN+1ビット幅の内容である．除数と被除数は，どちらも符号なしの数であると仮定している．わり算は次のように行われる．

```
(1)   If(除数 <= ウィンドウの内容)
          商はオーバーフローする(Nビットより大きくなる).
          *** ERROR ***

(2)   N回繰り返す:
      {
          If(商 <= ウィンドウの内容)
          {
              ウィンドウ = ウィンドウ - 除数;
              商の最右ビット = 1;
          }
          被除数と商を左に1ビットシフトする
      }
(3)   If(除数 <= ウィンドウの内容)
```

```
        {
            ウィンドウ = ウィンドウ - 除数 ;
            商の最右ビット = 1 ;
        }
(4)     STOP：ウィンドウは余りと商を保持している.
```

具体的な例をみれば，このさきは理解するのがもっと簡単である．6ビットの数（100010=34）を3ビットの数（110=6）でわり，3ビットの商（101=5）と3ビットの余り（100=4）をだす．4つはすべて符号なしである（先頭の1は負数を示すものではない）．ウィンドウは被除数の一部を囲む箱として示されている．この箱の左にある桁は捨てられる．

初期値：

ステップ1：
除数＞ウィンドウの内容；オーバーフローなし

ステップ2（反復1）：
除数＞ウィンドウの内容：ひき算をしない．
しかし，被除数と商を左にシフトする．

ステップ2（反復2）：
除数＜＝ウィンドウの内容：ウィンドウの内容から除数をひいて結果をウィンドウにおく．
（1000−110=0010）．商の下位ビットに1をおく．
商と被除数を1ビット左にシフトする．

ひき算………そして……………シフト：

ステップ2（反復3）：
除数＞ウィンドウ：ひき算しない．
しかし，商と被除数を左にシフトする．

2.5.4 切捨て，オーバーフロー，速度

乗算と除算に関係するおもな問題は，結果の正確さである．数を右へシフトすると，もっとも右にあった桁は切り捨てられる．続いて左にシフトしても元にもどらない．結果として，わり算（右シフト）にかけ算（左シフト）を続けると正確ではなくなる（わり算のとき大事な桁をいくつか切り捨ててしまうから）．もしかけ算をさきにすれば，正確さは失われないだろう．しかし，計算機のワード幅をオーバーフローするかもしれない（結果は有効なビットで表せる大きさを越えているかもしれない）．もちろん，積のワード幅が乗数や被乗数の2倍ならば，オーバーフローの問題はないだろう．しかし，これがいつもできるとは限らない（例えば，2つの **double** の数をかけなければならないときなど）．

関連した問題がいくつかある．例えば，−1/4は0ではなく−1になる．ここでは，−1は値0xffffである．4でのわり算は右シフトを行う．しかし，上位ビットはシフトの一部として拡張された符号である．つまり，−4/1はなにもしない（0xffffは2ビット右にシフトされても0xffffである）．問題によっては，使われているわり算アルゴリズムのため思わぬ結果になる．ウィンドウ・レジスタは演算数とはサイズが違うから，数の上位ビットが誤って切り捨てられることもある．例えば，49152/−1は16384となる．同じ数を16進数の表現でみると理由がわかる．すなわち，0xc000/−1==0x4000である．上位ビットは誤って切り捨てら

れている．

切捨てとオーバーフローの問題は，特に浮動小数演算で目立つ．そこでは，わり算とかけ算を間違った順序で行うと，正確さを失う．要は，オーバーフローの問題を心配しなければならないのはプログラマである．C 言語にはオーバーフローを処理する用意はない．どのような演算（加算や減算を含めて）を行う前にも，2つの演算数が範囲内にあるかチェックするのはプログラマの義務である．

かけ算とわり算のもうひとつの問題は，速度である．かけ算とわり算は演算に時間がかかる（シフトなどに比べて，時には，連続したたし算やひき算に比べても）．浮動小数演算は整数演算に比べてもっと時間がかかる．効率が重要ならば，計算についてよく考える必要がある．かけ算やわり算を最小限にするアルゴリズムを開発することは価値がある．

別の問題は，型変換に時間がかかることと，式の評価でしばしば暗黙の型変換を行うことである．例えば，**char** を含む式は，**int** だけを含む式の値を求めるよりも時間がかかる．自動型変換のルールは，すべての **char** を **int** にかえてしまうからである．同じ自動型変換のルールは，**float** と **double** にも適用される．つまり，すべての **float** は，計算される前に，**double** に変換される．その結果，**double** の2数より **float** の2数で同じ演算をするほうが（たとえ，**float** の方が短くても）時間がかかる．原則は，数を格納するのなら **char** よりも **int** に，**float** よりも **double** を使う．**char** に変数をおくことでメモリを節約しても，型変換をする余計なコードで相殺される．**float** と **double** を使うことでメモリの節約は実現するかもしれないが，**float** を使うと速度の問題が起こる．速度が重要ならば，最悪の場合の型にすべての数を格納することで型変換を最小限にする．いくつかの演算数が **double** ならば，（たとえ結果が整数になるとしても）すべて **double** に格納する．C は次々に生成されるローカル変数を再利用するから，最悪の場合でもメモリは浪費されない（4章で，この再利用がどのように行われているかをみる）．

2.6 ブール代数

2.6.1 論理演算子

C 言語はさまざまな論理演算子をサポートしている．そのうちのもっとも単純なものは，比較演算子である．A と B がどの式にもあるとき

```
A  >   B        はAがBより大きければ真である.
A  <   B        はAがBより小さければ真である.
A  >=  B        はAがBより大きいか等しければ真である.
A  <=  B        はAがBより小さいか等しければ真である.
A  ==  B        はAがBに等しければ真である.
A  !=  B        はAがBに等しくなければ真である.
```

“等しい”が2重の等号(==)であることに注意すべきである．等号ひとつのときは代入(a=bはbの内容をaに入れて，bの内容を評価する)を表す．この2つを混同しないように注意しなさい．

C言語では，数0は偽を示し，0でない数(負数も含む)は真を示す．論理演算子は，式の評価を数にして(真なら1，偽なら0)，他の演算子と同じように作用する．式

```
X + (A > B)
```

はまったく正当で問題ない．AがBより大きければ(X+1)に評価され，AがBより大きくなければ(X+0)に評価される．Cには3つのブール論理演算子，AND(&&)，OR(||)，NOT(!)がある．&と|が2重になっている．それで，ビットごとのAND演算子&やビットごとのOR演算子|と混同しないように注意すること．AND，OR，NOTは次の真理値表を使用する．

```
NOT !             AND &&                 OR ||

A | !A        A | B | A && B        A | B | A || B
---|----      ---|---|--------      ---|---|-------
F | 1         F | F |    0          F | F |    0
T | 0         F | T |    0          F | T |    1
              T | F |    0          T | F |    1
              T | T |    1          T | T |    1

T == 真（0でない）
F == 偽（0）
```

真理値表は，かけ算表のような働きをする．AとBは式中の2数である．もっとも右の欄は演算の結果を表し，AとBの規定された値を与える．真理値表のように書かれたAのかけ算表は次のようにみえるだろう．

```
A | B |  A * B
---|---|--------
1 | 1 |    1
2 | 3 |    6
4 | 5 |   20
   etc.
```

1*1==1，2*3==6，などである．同様に，T||F==1(左側の演算数が

ゼロでなく，右側の演算数がゼロならば，式全体は1と評価される)．AND，OR，NOTの真理値表は暗記する価値がある(A　&&　BはAとBがどちらも真のときだけ真であり，A｜｜BはAかB，または，AとBどちらも真のとき真であることを覚えればよい)．

&&と｜｜は他にない特性を持っている．&&と｜｜が使われている式中の評価の順序は左から右に行われることになっている（かっこで囲んで，順序をかえようとしていなければ)．そして，評価は式の真偽が決定されたとき終了することになっている．それで，式

```
if( x && spot( ) )
```

において，xが偽(0)ならばspot(　)は呼ばれない(なぜなら，xが偽だと式の評価は終了するから)．同様に

```
if( x || dog( ) )
```

では，dog(　)はxが真(1)ならば呼ばれない(なぜなら，サブルーチンの戻り値に関係なく式は真と評価されるから)．

2.7　簡単な恒等式

論理演算子は，よく制御ループやifに使われる．効率の点からいって，これらはしばしばプログラム中のもっとも危ない部分である（よく実行される部分である）から，できれば論理式を最適化することはよい考えである．Cコンパイラの中にはいくぶん最適化するものもあるが，あてにはできない．この章では式を最適化するさまざまな簡単な方法を示す．

2.7.0.1　式の反転とド・モルガンの法則

もっとも簡単な論理演算子は，NOT(！)である．！AはAが偽(ゼロ)なら，1になる．Aが真(ゼロ以外)ならば，0になる　！と他の演算子との組合せで巧妙なことができる．特に

```
!(A  ==  B)    ==    (A  !=  B)
!(A  !=  B)    ==    (A  ==  B)
!(A  >   B)    ==    (A  <=  B)
!(A  <   B)    ==    (A  >=  B)
!(A  >=  B)    ==    (A  <   B)
!(A  <=  B)    ==    (A  >   B)
```

である．最初の2つが同じであることはあきらかであるが，他は少し考えを要する．

演算子！は，AND や OR とも使われる．

```
!(A && B)   ==   (!A || !B)
!(A || B)   ==   (!A && !B)
```

この 2 つの恒等式は，ド・モルガンの法則として知られる．重要なことは（！A｜｜!B）がコンピュータへの 3 つの動作を表していることである．！A，！B，｜｜は，別々に計算されなければならない．一方で，！(A && B）は 2 つの動作（&& と！）だけである．それで，さきのと後のでは実行時間を 3 分の 1 減らすことができる．この問題のもう一方の側面は，読みやすさである．時には，プログラム全体を読みやすくするために，式を非能率的のままにしておくことも価値がある．場合によって，代数的変形を行うかどうか決めなくてはならない．

2.7.0.2　ブール代数の簡単化

ド・モルガンの法則に加えて，他にもいくつか役立つ恒等式がある．よく使われるものを図 2･9 に示す．

```
            式 OR:                       |          式 AND:
 ----------------------------------------|----------------------------------
1. !(!A)        ==  A                    |
                                         |
2. !(A ||  B) == (!A && !B)              | !(A &&  B) == (!A || !B)
3.  (A ||  0) ==  A                      |  (A &&  1) == A
4.  (A ||  1) ==  1                      |  (A &&  0) == 0
5.  (A ||  A) ==  A                      |  (A &&  A) == A
6.  (A || !A) ==  1                      |  (A && !A) == 0
                                         |
7.  (A||B)        == (B || A)            |  (A&&B)        == (B && A)
8. ((A||B) || C) == (A || (B || C))      | ((A&&B) && C) == (A && (B && C))
9. ((A||B) && C) == (A&&C) || (B&&C)     | ((A&&B) || C) == (A||C) && (B||C)
                                         |
10. (A && B)  || (A && !B) ==  A         | (A || B) && (A  || !B) ==  A
11.     A     || (A &&  B) ==  A         |     A    && (A  ||  B) ==  A
12.     B     || (A && !B) == (A || B)   |     B    && (A  || !B) == (A && B)
                                         |
```

図 2・9　恒等式

2.8　ビットごとの演算子とマスク

2.8.1　AND，OR，XOR，NOT

通常の論理演算子に加えて，C 言語はいくつかビットごとの演算子をサポートしている．ビットごとの演算子は，通常の演算子と同じ真理値表を使用する．し

かし，1度に1ビット（または，1カラム）ずつを対象に演算する．例えば，数101と011のビットごとのANDをすると，最右のビットどうしをANDして答えの最右のビットに結果（1 AND 1==1）をセットする．まん中の2つのビットのANDをとって，答えの中央のビットに結果（0 AND 1==0）をセットする．最左のビットも同じように処理する．いくつかの例を図2·10に示す．

Cには，4つのビットごとの演算子，&（AND），|（OR），~（NOT），^（XOR）がある．ビットごとのANDとビットごとのORは，通常のAND（&&）とOR（||）とは違い，ひとつの&と|が使われている．これらの演算子は，かなり違う動作をするので混同してはならない．ビットごとのNOT（~）は演算数の全ビットを反転させて（1を0に，0を1にする）評価する．しかし，演算数自身はかわらない（演算数が0でないとき0に，0のとき1に評価する論理演算子のNOTとは混同しないこと）．演算子XOR（排他的OR）は次の真理値表に従って結果を出す．

```
A | B | A ^ B
--|---|-------
0 | 0 |   0
0 | 1 |   1
1 | 0 |   1
1 | 1 |   0
```

図 2·10　ビットごとの演算

排他的ORは，対応するビットが異なる桁を真とする演算子である．演算数が異なれば結果は1になる．つまり，どれか一方だけが真ならば結果は真になる．

2.8.2　マ　ス　ク

ビットごとの演算子の，興味深い特質が図2·10ではあきらかである．下にある演算数は，答えを導くために，上にある演算数が処理されるフィルタのようにみえる．この下になっている演算数はマスクと呼ばれる．AND演算をみると，マスクに1があるところは，上にある演算数がそのまま答えまで通過している．マスクに0があるところは，答えは0になっている．ANDマスクはビットを選んでクリアしたり（0にセットしたり），あるビットだけを調べるのに使われる．例えば，式c=c & 0x7fは変数cの下位7ビット以外をクリアする（これは，特に入力文字からパリティビットを除くのに有効である）．式では

```
if( A & 0x03 )
        do_something( );
```

と表す．do_somethingはAの下位2ビットがセットされているときだけ実行される．

ORマスクも同じような働きをする．しかし，マスクで0になっているビットのところだけ，上の演算数が変更されないで答えまで通過する．ORマスクに1があると，答えの対応するビットはセットされる（1になる）．ORマスクは，ビットを個別にセットするのに使われる．例えば，x=x | 1はxの最下位ビットを1にする．

最後に，XORマスクはビットを選んで反転させるのに使われる．XORマスクで1があるところは，上の演算数の対応するビットだけが答えでは反転される（1が0にされ，0が1にされる）．XORマスクは，ビットマップド・ディスプレイのひとつのビットを反転させたいときなど，グラフィック・アプリケーションに有効である．また，ハードウエアのレジスタ中のひとつのビットを反転させるときにも使える（いいかえれば，ひとつを除いてレジスタの全ビットが，エラーのときは1に，成功のときは0にセットされる．その時，除かれたひとつのビットは，他のビットとは反対になっている．XORマスクは，その正反対のビットを反転して，成功のときはレジスタ全体が0であるとしてテストすることができる）．

2.9 コンマ，等号，条件演算子

C 言語でサポートされている演算子のうち2つは，解説をつけるに値するほどかわっている．それは，コンマ(,)と条件演算子(? :)である．さらに代入演算子(＝)は，たいていの言語にあるものとは異なる扱いをされている．

まず，相当(＝＝)と代入(＝)を混同しないように注意する．式

```
x == y
```

は，x や y を変更しない．式は，x と y が等しければ 1 になり，x と y が等しくなければ 0 になる．文

```
x = y
```

は x を y の内容に変更する．この2番目の式は，代入が行われた後の x の内容に評価される．

＝＝や＝の演算子と式を使って，何かを評価することに気がつくことは重要である(すべての式は何かを評価する)．代入演算子は，他の演算子のように演算子のひとつである．そして，やはり他の演算子と同じく，算術式の一部になることができる．例えば

```
x = 5 + (y = z);
```

は，完全に正しい文である．z の内容が y にコピーされる（かっこ内の式は，代入後 y の内容に評価される)．それから，y の内容が 5 に加えられて，結果が x におかれる．＝の優先順位が低いことに注意する．式

```
x = a + b = z + 5 ;
```

には，かっこが暗示されている．すなわち

```
x = (a + b) = (z + 5);
```

である．式に値を代入できないから，コンパイラはこれをみつけたらエラー・メッセージを出すであろう．

コンマは，C 言語の多くの記号と同様に，その文脈に依存するさまざまな機能を持っている．関数の引数どうしを分けることにも，ひとつの演算子としても使用される．この使い方を混同しないことが大切である．関数の引数並びにあるコンマは算術演算子ではない．コンマ演算子の主な目的は，**for** 文で複数の式を書かせることである．例えば

```
for( i = 0, j = size ;  i <= j ; i++, j-- )
```

のループでは，i と j が **for** 文の左の部分で初期化され，どちらも右の部分で更新されている．

for ループの目的は，1ヶ所にループの制御を集中させることであるから，**for** 文の中にループ制御を処理する変数を持たない演算を含んでいるのは，悪い書き方だと考えられる．コンマ演算子は，このようによく乱用される．つまり，そこにあるべき仕事を持たない式を for 文において使う（なぜなら，ループの制御を扱う変数を持たないから）．例えば

```
for( x += 7, y = 1; y < 10; y++ )
        body( );
```

では，x を含んでいる文は，制御に使われていないから，ループには属さない．

この文は次のように書き直されるべきである．

```
x += 7;

for( y = 1; y < 10; y++ )
        body( )
```

コンマ演算子は，実際にはどんな式にも使うことのできる演算子である．コンマ演算子を，代入をしない，代入演算子としてみることができる．つまり，式

```
x = ((y += 1), (z += 1));
```

は，x と y はどちらも 1 増加する．文は，代入演算子（＝）に類以したやり方で，コンマの右にある式の内容に評価される．つまり，（増加後の）z の値が x に加えられる．ここで，優先順位に注意すべきである．コンマはどの演算子（代入演算子も含む）よりも低い優先度を持つ．例えば

```
x = ( 1, 2 );
```

はコンマ演算子で 2 に評価される．式

```
x = 1 , 2;
```

は 1 に評価される．しかし，コンパイラのかっこ付けは次のようになる．

```
(x = 1) , 2;
```

コンマ演算子は，場合によってはマクロに役立つ．しかし，それは読みにくいので避けるべきである．前述の式は次のように書かれるべきである．

```
z += 1;
x = (y += 1);
```

3 番目の興味深い演算子は条件演算子（？：）である．条件演算子は 3 項演算子

である．すなわち，2つではなく，3つの部分を持つ．

```
x + y
```

は2項演算子が2つの部分を持ち，ひとつの結果（和）に評価される．3項演算子

```
x ? y : z
```

は3つの部分を持つ．しかし，やはりひとつの結果に評価される．xが真ならばyに評価され，xが偽ならばzに評価される．次のように，どちらの項もサブルーチンであるとする．

```
x ? sam( ) : dave( )
```

この時は，どちらか一方のサブルーチンだけが呼ばれる．条件式は，ひとつの演算子である．それで，次の式にも使われる．

```
a = x ? y : z;
```

この式は，xの真偽によって，yかzの内容をaに代入する．この例では，条件式を等価の

```
if( x )
        a = y;
else
        a = z;
```

にするほうが好ましい．なぜなら，条件式から生成されたコードが，もっと効率的である（アドレスが，2度でなく1度計算されるだけでよい）．

条件式は，他のところにも使用できる．例えば

```
printf( "x is %s", x ? "TRUE" : "FALSE" );
```

は，xがゼロでなければ“x is TRUE”を書き，xがゼロならば“x is FALSE”を書く．ここで，条件式は文字列のどちらか一方に評価される．等価のコード

```
if( x )
        printf( "x is TRUE" );
else
        printf( "x is FALSE" );
```

は，かえって効率的でない．ここでは2つの文字列に多くの文字を格納する必要がある．さらに重要なことは，サブルーチンを呼ぶプログラムをひとつでなく2つつくる必要がある．

条件演算子は入れ子にすることもできる．例えば

```
#define YES     0
#define NO      1
#define MAYBE   2
```

```
printf("x = %s",  x == YES  ?  "YES"   :
                  x == NO   ?  "NO"    :
                               "MAYBE );
```

評価の順序をあきらかにするために，入れ子になった条件式を上のように書くことは大切である．

```
printf("x = %s",  x == YES ? "YES" : x == NO ? "NO" : "MAYBE" );
```

はさきの例よりも読みにくい．

2.10　バ　　　グ

予想しない符号の拡張や，数の切捨ての結果，浮かび上がってくるバグがいくつかある．例えば，**char** はたいていのコンパイラでは符号付きの数である（しかし，**unsigned char** として宣言できる）．すべての **ASCII 文字**は，数の下位７ビットで表される．しかし，計算機の中には特殊なキーを８ビットで表しているものもある（IBM-PC のように）．例えば，IBM のキーボード上の補助キーをたたくと，ROM-BIOS は２バイトコードをソフトウエアに渡す．最初のバイトはいつも０で，第２バイトは ASCCI コードではない特殊なコードである１バイト入力のルーチンに２バイトかえってくると不都合であるから，第２バイトはしばしば，特殊なコードであることを示すために，最上位ビットを１にセットしてかえされる．F1 キーをたたくと，２文字シーケンス 0x00　0x3b（２進数では 00000000　00111011）を送り，それから 0xbb（10111011）にマップされる．

このマッピングは，次のサブルーチン getkb(　) で行われる．getkb(　) は，ハードウエアから１文字得るために scan(　) をよび，２文字シーケンスを１文字にマップする．

```
getkb( )
{
        char x;

        if( (x = scan( )) == 0 )
                x = scan( ) | 0x80 ;

        return x;
}
```

getkb(　) は，ほとんどの文字入力ルーチンが行うように，**int** をかえす．どの式でも，**char** の変数はすべて式が評価される前に **int** に変換される．予約語 **return** は，式を引数とする．それで，x は式であるかのように扱われ，型変換が行われ

る．x が 0xbb を含んでいるとすると，この数は，**char** が **int** に変換されるとき，0xbb が 0xffbb に変換される（0xbb は負数である．符号拡張が行われて 0xffbb になる）．したがって

```
if( getkb( ) == 0xbb )
```

でテストしてもうまくいかない．なぜなら，0xbb でなく 0xffbb が getkb( ) からかえされる．この問題は，次のように訂正することができる．

```
if( (getkb( ) & 0xff) == 0xbb )
```

または，getkb( ) の中で

```
return( x & 0xff );
```

としても動くだろう．AND 演算が行われる前に x が **int** に変換されるだけである．

関連した問題が，通信を行うハードウェアとのインタフェースにある．このハードウェアの大部分が，パリティ・チェックと呼ばれるエラー検知機構をサポートしている．偶数パリティが使われていれば，すべての文字は，偶数個のビットが 1 にセットされる（2 進数 101 は偶数個，この場合は 2 つビットがセットされている．010 は奇数個のビットがセットされている）．ASCII コード全部が偶数個のビットが 1 になっているのではないから，パリティ・ビットが数を満たすために使われる．それで，偶数個の 1 のビットを持つことになる．例えば，ASCII コードの a の数値は 0x61 である（2 進数で 01100001）．この数は奇数個のビットが 1 である．したがって，ASCII コードの a が，偶数パリティを使用してモデムを通して送られるときは，最上位ビットが 1 にセットされ，この数は偶数個の 1 のビットを含むだろう．つまり，数 0x61（01100001）は 0xe1（11100001）に変換される．それは 8 ビット符号付きの **char** で表した数である．この文字は直接ハードウェアから読むことができる．それから，**int** の変数に格納され，符号の拡張が行われる．つまり，0xe1（11100001）が，0xffe1（1111111111100001）に変換される．

古いコンパイラの中には，Lattice C のように，暗黙のうちにすべての **char** を符号なしとして扱うものもある．したがって，この符号拡張の問題は，**char** を符号付きの数であるとする，比較的新しいコンパイラにプログラムを移植するときまで現れないだろう．

自動型変換と符号拡張は，ともに，他にも予想外の方法で行われることがある．

例えば，8 ビットの **int** を使用した式

```
(64 * 2) / 2
```

は −64 と評価される．64 は 16 進数で 0x40 である．それに 2 をかけると，0x80 になる．しかし，0x80 は 8 ビットの計算機では負数である（値は −128 である）．その数を 2 でわると 0xc0 になる（符号拡張をともなった右シフトをする）．すなわち −64 である．

オーバーフローもまた問題である．

```
(64 * 32) / 16
```

は，8 ビットの **int** ならば，0 に評価される．32 でかけることは 5 ビット左シフトすることである．しかし，64 を 5 ビット左にシフトすると，唯一の大切なビットを数の左端に消してしまう．つまり，01000000 は 5 ビット左にシフトすると 0100000000000 になるが，8 ビットの精度では下位 8 ビット以外は，すべて切り捨てられ 0 になる．

これらの問題は 10 章で再び取り上げる．

2.11　練　　　習

2-1　10 進数 93 と −56 を 2 進数，16 進数，8 進数に変換しなさい．8 ビット語長を使用すること．

2-2　10 進数 17 と −11 を 2 進数にして，それらを 2 進数でかけ，その過程を（図 2·7 のように）示しなさい．6 ビットの演算数を使って，12 ビットの結果を生成しなさい．

2-3　2 進数で表した 21 を 3 でわり，その処理の過程を示しなさい．6 ビットの符号なし非除数と 3 ビットの符号なし除数と商を想定しなさい．余りは何になるか．

2-4　次の式をどのように評価するか．その作業の過程を示し，答えを 10 進数と 16 進数で表しなさい．すべての演算数と答えは 8 ビットの数を想定しなさい．C の構文を数の基数を識別するために使いなさい．

```
a.  (111 & 011) ^ 0xa5
b.  -(0x55 + 0252)
c.  127 * 3
d.  (0x62 << 1) / 2
e.  17 | 36
f.  64 * (32 / 16)
g.  (17 < 11) + ((93 && 66) || 56)
```

2-5 次の式は何を行うか．2，3の具体的な例をあげて答えを証明しなさい．

a ^= b ^= a ^= b;

2-6 ときには，**int** や **long** の範囲より大きな数を操作する必要がある．これらの大きな数の計算を行うことができる large integer の操作パッケージがいくつか存在する．数自体が配列に入れられている．次の large integer ルーチンを書きなさい．

```
#define NUMBYTES 8

typedef char     BIGINT[ NUMBYTES ]    /* 64 ビット幅の integer */

b_atoi( str, num )
char    str[ ];
BIGINT  num;

mult(  multiplicand, multiplier, product )
BIGINT multiplicand, multiplier, product;

add(    num1, num2, sum )
BIGINT  num1, num2, sum ;

negate( num );
BIGINT  num;

print_hex( num );
BIGINT     num;
```

b_atoi（str，num）は large integer の16進表現を含む ASCII の文字列を入力する．そして，その数を2進数に変換し，num に2進数で格納する．例えば

```
BIGINT  num;
b_atoi( "0x123456789a", num );
```

への呼出しは，次のように配列 num を初期化する．

```
num[0] == 0x9a   <-- LSB
num[1] == 0x78
num[2] == 0x56
num[3] == 0x34
num[4] == 0x12
num[5] == 0x00
num[6l == 0x00
num[7] == 0x00
```

mult（multiplicand，multiplier，product）は，multiplier に multiplicand をかけ，product に結果をおく．結果が大きすぎて，BIGINT に格納できなければ0をかえし，格納できれば1をかえす．**add（num1，num2，sum）**は，

num1 と num2 を加えて結果を sum におく．**negate (num)** は2の補数を使用して，num を負数にする **print_h (num)**；は数を16進でプリントする．プログラムは，ワード幅（NUMBYTES として **#define** しておく）を NUMBYTES だけを修正するだけで変更できるようにするべきである．つまり，ワード幅を変更するためには，NUMBYTES を修正して再コンパイルするだけでよい．この場合プログラム自体は変更する必要がない．プログラムは NUMBYTES で（無理のない範囲の）正の整数を設定して動くようにすること．

ルーチン全体は負数や0の非演算数を適切に処理することができること．さらに，2つの非演算数は，演算数のうちひとつが結果（sum，product 等）になるのでなければ，変更できない．mult( )，add( ) への3つの引数はすべて同じでよい．例えば

```
BIGINT  num;
b_atoi( "10", num );

add( num, num, num );
```

で動作する．add( ) からかえったとき，num は20を含んでいる．自分でテスト・データを工夫しなさい．しかし，次の数はデータに含めなさい．

```
n1      = 12345678         (10進数)
n1      = 0000000000bc614e (16進数)
n2      = -87654321        (10進数)
n2      = ffffffffac6804f  (16進数)
-n1     = fffffffff439eb2
n1 + n2 = ffffffffb82e19d
n1 - n2 = 0000000005f5e0ff
n1 * n2 = fffc27c9d91d0712
n1 / n2 = 0000000000000000 (R 0000000000bc614e)

n1      = 12345678901234567 8 (10進数)
n1      = 01b69b4ba630f34e   (16進数)
n2      = 123456789012345678 (10進数)
n2      = 01b69b4ba630f34e   (16進数)
-n1     = fe4964b459cf0cb2
n1 + n2 = 036d36974c61e69c
n1 - n2 = 0000000000000000
n1 * n2 = (無効)
n1 / n2 = 0000000000000001 (R 0000000000000000)
```

2-7　次のルーチン

```
div( dividend, divisor, quotient, remainder )
BIGINT dividend, divisor, quotient, remainder)
```

を 2-6 で作成した BIGINT パッケージに加えなさい．このルーチンは，dividend を devisor でわり，結果の数を商と余りに入れる．作成するルーチンは次の式

```
12345678901234567 8/12345678901234567 8
```

を計算し，商が 1，余りが 0 にならなければいけない．余りが正しい符号になっているか確かめること．次の式が維持されているべきである．

```
((dividend/divisor) * divisor) + remainder == dividend
```

2-8　b_atoi( ) を変更して，10 進数と 16 進数で表した文字列を入力とすることができるようにしなさい．0x が前におかれているか調べ，あれば 16 進数として扱い，なければ 10 進数として扱うようにすること．BIGINT num を 10 進数で出力するルーチン **print_d**（**num**）を書きなさい．

3　アセンブリ言語

Cプログラマが，アセンブリ言語の基礎を知る必要があるのには，いくつか理由がある．高級言語ではあるけれども，Cはかなりアセンブリ言語に近い．通常，アセンブリ言語で書かれる多くのタスク（高機能算術計算ルーチン，割込みルーチン，等）は，C言語でも書くことができる．C言語を活用するためには，アセンブリ言語の基礎知識を知っている必要がある．同様に，C言語の演算子の多く(ビットごとの論理演算子，シフト演算子，等）は，アセンブリ言語命令に似ている．アセンブリ言語を知ることは，C言語を理解する助けにもなる．多くのアセンブリ言語の概念(間接命令，等)が，C言語の概念(ポインタ等)に反映している．基礎になる言語を学ぶことは，より高度な構成概念を理解するのに役立つ．しかし，アセンブリ言語を知ることの，もっとも重要な理由はデバッグである．次章で，Cコンパイラで使用されるコード生成技術について詳しく調べる．コンパイラがどのようにサブルーチンを呼ぶのか，変数がどのようにメモリに格納されるのかを知ることは，発見しにくいバグを徹底的に調べるときにたいへん役立つ．アセンブリ言語を少しも知らないでは，そんなバグを解決できない．

本章では，プログラマの立場から，コンピュータがどのように構成されているか，そのコンピュータを操作するために必要とされる基本的なアセンブリ言語をみる．ここで述べる言語と計算機は実在しない(もっとも，68000やPDP-11をかなり引用している)．このデモ計算機は，たいていの実在の計算機よりプログラムするのが簡単である．実際のアセンブリ言語プログラミングに共通する，多くの問題にもここでは触れていない．さらに，この言語は，C言語に関連した問題をみるのに役立つように設計されている．概念を学ぶのであるから，不必要な詳細

を述べて複雑にすると的はずれになる．

人は，アセンブリ言語にいくぶん好き嫌いがある．すでに8080かZ-80を知っていて，他のものを知らなければ，アドレッシング・モードのセクションを特に注意して読むようにする．デモ計算機は，拡張アドレッシング・モードはサポートしていない．本章での，もうひとつの重要な概念は“サブルーチン・コールとスタック”のセクション（3.5.4）にある．これは，次章に進む前によく理解しておく必要がある．

3.1　アセンブラとは何か

アセンブリ言語（あるいはアセンブラ）は，その各命令が，直接機械レベルのコードに翻訳されるプログラミング言語である．つまり，アセンブリ言語のプログラムは，計算機が使う2進コードに直接変換することができる．ひとつのアセンブリ言語命令は，直接ひとつの機械命令に変換される．このレベルでは，コンピュータは基本的に精巧な加算機にすぎない．それは，ひとつのメモリ・セルの内容を他へ転送したり，2つの数をたしたりするような単純な動作だけができる．多くの卓上計算機は，たいていのコンピュータよりももっと強力な命令を持っている（ただし，コンピュータの方がはやい）．C言語のような高級言語は，数行のアセンブリ言語命令を必要とする動作を1行で行うこともできる．実際には，コンパイラを高級言語からアセンブリ言語への変換機としてみることもできる．

3.2　メモリ構成とアライメント

コンピュータのメモリは，それぞれが別のアドレスを持ったセルの並びからできている．よく似ているものは，ページごとに数が付いた本である．ページの物理的なサイズは，1ページに書いてある行数で制限される．行数が，1ページ当りの行数より大きくなったら，次のページにしなければならない．このたとえでは，各ページが1つのセルであり，ページ数がセルのアドレスである．1ページにちょうどよい行数は，計算機のワード幅（メモリのひとつのセルに含まれる2進数の桁の数）である．もっと厳しい制限がある．2ページ使うほど大きな数は，向かい合う2ページを使わなければならない．大きな数の半分に対して，1ページを当てはめることはできない．この最後の制限は，アライメント（alignment）と呼ばれる．本の最初のページは，0（0は偶数である）と数字が付けられる．実際に，

左側のすべてのページは偶数である．ここで使う計算機は，16 ビットのワードである．各ワードは，2 つの 8 ビットのバイトに分割される．そして，各バイトにはアドレスがある．つまり 1 ワードには 2 バイト必要であるから，各ワードは 2 つのアドレスを占める．このデモ計算機は，バイトとワードの操作が異なる．バイト操作は，計算機内のどのバイトに対しても行うことができる．しかし，ワード操作（16 ビットの加算のような）では，いつも演算数の下位バイトに偶数アドレスを使う．そして，その上位バイトには，下位バイトより 1 つ大きいアドレスを使う．ワードが偶数アドレス上に並ぶことになるので，これをワード・アライメントと呼ぶ．計算機の中には，もっと複雑なアライメント制限をさせるものもある．例えば，8086 のセグメントレジスタは，16 バイト（paragraph）毎の境界線（boundary）上に配置しなければならない．

デモ計算機のアライメントを図 3･1 に示している．MSB は最上位バイト（Most Significant Byte）を表し，LSB は最下位バイト（Least Significant Byte）を表す．10 進数で説明すると，2 桁の数 56 では 5 が最上位バイトで，6 が最下位バイトである．

```
                 ビットポジション

                15      8 7       0
                |--------|--------|
アドレス     1  |  MSB   |  LSB   |  0
                |--------|--------|
             3  |  MSB   |  LSB   |  2
                |--------|--------|
             5  |  MSB   |  LSB   |  4
                |--------|--------|
                |      etc.       |
```

図 3・1　メモリ構成

本章にあるすべての命令は，W が続いていなければ，1 バイトに対して操作を行う．例えば，MOV 命令は，1 バイトをある場所から別の場所へ移す．MOV.W はワードサイズのもの（2 バイト）を転送する．

コードとデータは，同じメモリ領域を共有する．コンピュータは，セルに命令が入っているのか，データが入っているのか区別できない．たいていのマイクロコンピュータでは，C プログラムが実行中のコードを誤って修正することが可能である．実行中であれば必ず破滅的なことになる．また，データのメモリ領域を誤って実行し始めることもある．これも悲劇的なことになる．

3.3　レジスタと計算機の構成

アセンブリ言語がどのように動くか理解するために，まず，それが動作する計算機についてみてみなければならない．たいていのコンピュータは，2つのはっきりと異なる部分―中央処理装置（CPU）とメモリ―に分けることができる．CPUは，バスと呼ばれる配線の集合を通して，メモリにアクセスすることができる．主メモリに加えて，すべてのCPUはレジスタと呼ばれる少数の内部メモリを持っている．ハードウェアによるCPU上のさまざまな拘束のために，CPUが主メモリにアクセスするのは，内部メモリにアクセスするよりも時間がかかる．したがって，よいCコンパイラは，生成するコードにかなりレジスタを使用している．C言語の予約語**register**は，レジスタが使えれば，変数の格納にレジスタを使わせる（レジスタの数に制限がある．コンパイラは，コンパイル作業にレジスタのいくつかを使う）．メモリのアドレスは，実際には，CPUがメモリにアクセスするとき，バスにかける1セットの電圧を表す方法である．CPUが，レジスタをアクセスするのにバスは必要ない．それでレジスタにはアドレスはない．代わりに，レジスタは名前を持っている．

簡単な計算機は，アキュームレイタ（accumulator）と呼ばれるレジスタ（卓上計算機では表面にある小さな窓にその内容が表示される）を持っている．すべての演算の結果は，そこで計算される．全部の命令がアキュームレイタを使う．たいていの計算機は，レジスタが2〜3個付加されている．レジスタには，アキュームレイタの現在の内容が格納され，後で取り出される．また，ある命令（格納や回復）にも使われる．コンピュータも，同じような構造を持っている．しかし，レジスタには，アキュームレイタより，A，B，C，等の名前が付けられている．同様に，命令はレジスタだけをアクセスするものがあり，一方，メモリをアクセスするものもある．

デモ計算機にはA，B，C，D，SP，PC，FP，FLと名付けられた全部で8個のレジスタがある．それらを図3·2に示す．

全レジスタは，16ビット幅（2バイト）である．A，B，C，Dレジスタは汎用レジスタである．これらはアキュームレイタとしても使える．他のレジスタは，専用レジスタである．これらのレジスタは，レジスタに影響する命令の説明をするときにどのように使用されるかわかるだろう（ただし，FPは次章で説明する）．

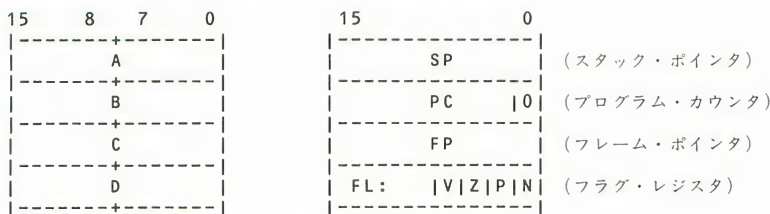



図 3・2 デモ計算機 レジスタ・セット

プログラム・カウンタ (PC) は特殊レジスタである. PCは, いつも現在実行中の命令のアドレスを保持している. PCは, 命令が実行されるごとに, 次に実行される命令のアドレスを保持するため, 自動的に更新される. デモ計算機の全命令は, 必ず2バイト占める. それでPCは通常, 命令ごとに2ずつ増加する. ここで, 重要な概念は, PCが命令そのものではなく, 次の命令のアドレスを保持することである. 命令が実行されるとき, 計算機は, まず, PCに示されたアドレスにある命令をフェッチして, その命令を実行する.

3.4 アドレッシング・モード

計算機の命令はすべて, その実行の途中でメモリかレジスタをアクセスする. 多くの実在するコンピュータとは違って, すべての命令がレジスタのいずれかを使う. 例えば, ADD命令は, 8個のレジスタのうちいずれかに数をたす. 多くのコンピュータは, 各命令が使うレジスタを制限している. 例えば, 卓上計算機のようなものでは, + キーはアキュームレイタだけを使う (他の格納用レジスタを使うことはできない). 格納用レジスタの内容を更新するには, アキュームレイタにその内容を移して, それからアキュームレイタの内容を更新し, 再び結果を格納用レジスタにもどさなければならない.

レジスタとメモリのアクセスは, 数種類の方法のうちのひとつで行われる. 例えば, メモリやレジスタを直接アクセスすることができる. また, あるアドレスからのオフセットにあるメモリをアクセスすることもできる. アドレスを使用するさまざまな方法をアドレッシング・モードという.

この後に続く命令の説明では, アドレス, または, レジスタを表すのにrNを使う. アドレスとレジスタは, さまざまなアドレッシング・モードで使用される

から，実際にはすべての場合を列挙しない．むしろ，rN の表記を後に続く物で置き換えることができる．

3.4.1 レジスタ直接アドレス

rN をレジスタ名で置き換え，そのレジスタの内容を参照する．このモードを，レジスタ直接アドレッシングという．命令 MOV A, B は，A レジスタの内容を B レジスタに移す．B レジスタの前の内容は壊され，A レジスタの内容はかわらない．これは 1 バイトの転送だから（.W が MOV についていない），A と B レジスタの下位バイトだけが使用される．上位バイト（ビット 8 から 15）はかわらない．MOV.W A,B は上位と下位のバイトが転送される．

C 言語での同じ操作は x=y である．ここで，x と y はレジスタ変数である．すなわち

```
register  int   x,y;
x = y;
```

である．

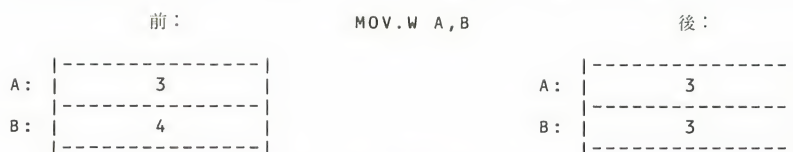



図 3・3　レジスタ直接アドレッシング

3.4.2 メモリ直接アドレッシング

rN が数に置き換えられると，その数をアドレスとするメモリの内容がアクセスされる．命令 MOV 100, 200 は，100 番地にあるメモリの内容を 200 番地にあるメモリに移す．100 番地の内容はかわらない，しかし，200 番地の内容は破壊される．ワード幅の命令 MOV.W 100, 200 は，100 番地のメモリの内容を 200 番地のメモリに移し，101 のメモリの内容を 201 のメモリに移す．MOV,W 100,A は，100 番地の内容を A レジスタの下位バイトに入れ，101 番地の内容を A レジスタの上位バイトに入れる．ワード幅のものを奇数のアドレスへ，または，奇数のアドレスから転送しようとするとエラーになる（つまり，MOV.W 101,A は間違っ

```
          前:              MOV.W 100,200              後:

         |---------------|                    |---------------|
  100:   |      999      |             100:   |      999      |
         |---------------|                    |---------------|
  102:   |       .       |             102:   |       .       |
                 .                                    .
  198:   |       .       |             198:   |       .       |
         |---------------|                    |---------------|
  200:   |       0       |             200:   |      999      |
         |---------------|                    |---------------|
```

図 3・4 直接アドレッシング

た命令である).

C 言語では，やはり x=y; であるが，2つの変数は static で宣言される．

```
static  int     x,y;
x = y;
```

メモリ・アクセスは，レジスタ・アクセスより時間がかかることを覚えておくべきである．同様に，たいていの計算機では，メモリ・アクセスはもっと長い命令長を必要とする(例えば，1ワードでなく2ワードの命令である).

3.4.3 即値アドレッシング

数の前に＃が付いていれば，指定されたアドレスのセルの内容でなく，その数自体が使われる．命令 MOV ＃100, 200 は，数 100 をメモリ 200 番地に入れる．MOV.W ＃999,A は，A レジスタに 999 を入れる．レジスタへの即値命令は，メモリを変更する命令よりもはやい．さらに，すべての即値命令はメモリからメモリへの転送命令よりもはやい．次は，C 言語での等価の文である．

```
register int  x;   /* A レジスタを使用する       */
static   int  y;   /* メモリ 200 番地を使用する */

x = '*';           /* MOV.W #42,A を生成する  */
y =  3 ;           /* MOV.W #3,200 を生成する */
```

最初の代入の 42 は，'*'のことである．C 言語では，クオートひとつ(')で囲まれた文字はその文字の ASCII コードに評価される．'*'は ASCII で 0x2a，または，10 進数の 42 である．

3.4.4 間接アドレッシング

表記(rN)は，間接アドレッシングのために使われる．このモードでは，指定

Aレジスタ内のアドレスにあるメモリ・セルの
内容をBレジスタに移動する.

図 3・5 レジスタ間接アドレッシング

されたレジスタ，または指定されたメモリ内のアドレスにあるメモリ・セルの内容がアクセスされる．例えば，レジスタAが100を含んでいると，命令MOV (A)，Bはメモリ番地100の内容をレジスタBに移す．この例が図3･5に示されている．

他の例では，レジスタAが100にセットされ，レジスタBが200を含んでいるとき，MOV (A),(B)はメモリ100番地の内容をメモリ200番地に移す．どちらのレジスタも変更されない．

最後の例は，命令MOV.W (100),(200)は，100番地の中にあるアドレスで指定されたセルの内容を，200番地の中にあるアドレスで指定されたセルに移す．これはワード幅で転送する．この例を図3･6に示す．

間接アドレッシングは，重要な概念である．レジスタが操作されるべきメモリのアドレスを含んでいる．そして，レジスタ自体は変更されない．レジスタが含んでいるアドレスにあるセルが操作対象である．この関係を表現するために，指し示すという言葉を使う．いまの例では，Aレジスタは，100番地のメモリを指し示している．つまり，100番地のメモリはアドレスが100で，Aレジスタに指

```
                 MOV.W (100),(200)

         前:                                  後:

98:  |                |              98:  |                |
     |----------------|                   |----------------|
100: |      304       |              100: |      304       |
     |----------------|                   |----------------|
102: |       .        |              102: |       .        |
             .                                    .
             .                                    .
108: |       .        |              108: |       .        |
     |----------------|                   |----------------|
200: |      302       |              200: |      302       |
     |----------------|                   |----------------|
202: |       .        |              202: |       .        |
             .                                    .
             .                                    .
300: |       .        |              300: |       .        |
     |----------------|                   |----------------|
302: |       0        |              302: |      999       |
     |----------------|                   |----------------|
304: |      999       |              304: |      999       |
     |----------------|                   |----------------|
306: |       .        |              306: |       .        |
```

100番地の中のアドレスにあるセルの内容を
200番地の中のアドレスにあるセルに移す.

図 3・6 メモリ間接アドレッシング

し示されている.

Cは，すべてのauto変数（staticと，はっきり宣言されていないローカル変数）をアクセスするために，間接アドレッシングを使う．次章でその理由を述べる．間接アドレッシングは，ポインタ操作にも使われている．例えば

```
register int    *p;     /* Aレジスタを使用する          */
int             x;      /* メモリ番地200を使用する      */
int             y;      /* メモリ番地202を使用する      */

p = &x;                 /* xのアドレスをpに入れる       */
                        /* MOV.W #200,A を生成する      */

*p = 5;                 /* 5をメモリ200番地に入れる命令 */
                        /* MOV.W #5,(A) を生成する      */

y = *p;                 /* MOV.W (A),202 を生成する     */
                        /* メモリ番地200と201の内容を   */
                        /* 202と203番地にコピーする     */
```

のように使う.

3.4.5　後置きの自動インクリメント付間接アドレッシング

このモードでは，表記(rN)＋が使用される．動作の実行は，アクセスされるセルのアドレスを，rNが保持している直接アドレッシングに似ている．しかし，rNに含まれているアドレスにあるセルが，アクセスされた後，レジスタの内容が更新される．命令がバイト・サイズのオブジェクトの操作であれば，1がrNに加えられる．この処理は，Cの＋＋演算子と同じである．例をあげると，Aが100を保持しているとき，命令MOV (A)+,Bは，100番地のメモリの内容をレジスタBに転送する．そしてレジスタAは自動的に101に増加される(この命令は，バイトを転送したのであるから)．AレジスタとBレジスタは，変更された．しかし，100番地のメモリは変わらない．MOV.W (A)+,Bは，100と101のセルの内容をBレジスタに移す．そして，Aレジスタに2が加えられる．ここで重要なことは，転送されるものの型によって増加する幅が違うことである．1バイト転送されると，レジスタは1増加する．1ワード転送されると，レジスタは2増加する．

C言語の自動インクリメントの，主な使用法は，ポインタを使った配列のアクセスである．

Aレジスタ内のアドレスにあるメモリ・セルの内容をBレジスタに移す．Bレジスタをワードのサイズだけ増加する．

図 3・7　自動前置きインクリメント付き間接アドレッシング

```
static   int    x[20] ;  /*  メモリ番地 200—239 を使用する  */
register int    *xp   ;  /*  A レジスタを使用する           */

xp = x;          /*     MOV.W  #200,A            */
*xp++ = 'c';     /*     MOV.W  #99,(A)+          */
```

この例は複雑なので，少し説明を必要とする（この題材は 7，8 章で詳述する．それで本章を読み終わるまでとばしてもよい）．x は整数配列であり，**int** は 2 バイト使用する．それで，この配列には 40 バイトのメモリを必要とする．配列の最初のセル（x[0]）は，200 と 201 番地にある．次のセルは，202 と 203 番地にある．残りも同様である．C 言語では，自動的に配列名がその配列の最初の要素のアドレス（200）に評価される．式 xp＝x は，配列を指し示すポインタ（この例では A レジスタ）を初期化する．つまり，A レジスタに x の最初の要素のアドレスが入れられる．

＊xp＋＋＝'c' を説明しよう．A レジスタは，200（x のアドレス）を含んでいる．間接アドレッシングが使われているから，アドレス 200 のセルの内容は変更される．それで 99（'c' の ASCII コード）がメモリの番地 200 と 201 に入れられる（これはワード転送だから）．それから，ポインタ（A レジスタ）が増加する．ワード転送だから 2 が加えられる．A は，202（配列の次の要素のアドレス）を含むことになる．面白いことに，ほとんど同じプログラムが文字配列に使われている．

```
static   char   x[20] ;  /*  メモリ番地 200—219 を使用する  */
register char   *xp   ;  /*  A レジスタを使用する           */

xp = x;          /*     MOV.W  #200,A            */
*xp++ = 'c';     /*     MOV    #99,(A)+          */
```

MOV.W が，やはりポインタを初期化するのに必要とされることに注意しなさい．すべてのポインタは，指し示すものに関係なく，同じサイズである（ポインタは，他の変数のアドレスを保持している変数である．だから，ポインタのサイズは，個々の計算機でアドレスを保持するのに必要なバイト数である）．文字配列を扱っているので（この配列は 20 バイトを占める），配列をアクセスするのには MOV.W よりむしろ MOV を使う．そして，ポインタには，2 ではなく，1 が加えられる（ワード転送ではなくバイト転送を使っているから）．MOV が実行された後，メモリ番地 200 は 99 を含み，A レジスタは 201 を含む．

3.4.6 自動前置きデクリメント付き間接アドレッシング

このモードでは，－(rN) 表記が使われる．動作は，フェッチする前にレジス

Aレジスタを減少させる．更新されたAレジスタの中のアドレスにあるメモリ・セルの内容をBレジスタに移す．

図 3・8　自動前置きデクリメント付き間接アドレッシング

タの内容をデクリメント（減少）すること以外，自動後置きインクリメント付と同じである．例えば，Aレジスタが100を保持しているとき，命令MOV −(A),BはレジスタAを99に減少し，そしてメモリ番地99の内容をレジスタBに移す．自動前置きデクリメントでも，レジスタを更新する数は転送されるもののサイズによって異なる．レジスタAが100を保持しているとすると，MOV.W −(A),BはAレジスタを2減少させて（ワード転送だから）98にする．そして，98と99番地のメモリ・セルの内容がBレジスタの下位と上位バイトに移される．さきほどの，Cプログラムの例を使って変更すると，前置きデクリメントは次のように使われる．

```
static   char   x[20] ;  /*  メモリ番地200—219を使用する  */
register char   *xp   ;  /*  Aレジスタを使用する          */

xp = &x[20];     /*     MOV.W   #220,A          */
*--xp = 'c';     /*     MOV     #99,-(A)        */
```

配列の最後を指し示すように，ポインタを初期化していることに注意しなさい．

実際は，配列の終わりをこえたひとつのセルを指し示すように初期化されている．セルをアクセスする前に減少させるのであるから，A レジスタは代入が行われるときには x[19]（のアドレスを保持して）を指し示す．

デモ計算機は，前置きデクリメントと後置きインクリメントをサポートしている．しかし，後置きデクリメントと前置きインクリメントはない．多くの実在するコンピュータでもそうである．たいていは，スタックの管理に前置きデクリメントと後置きインクリメントを必要とするが，後置きデクリメントと前置きインクリメントは必要としないからである．後でもっと深くこの問題を取り上げる．C 言語での関連した問題は，ひとつではなく 2 つの命令が，これらのサポートされていない関数のうちひとつを行うために必要とされる．例えば

```
xp = &x[20];     /*  MOV.W   #220,A                */
*xp-- = 'c';     /*  MOV     #99,(A)               */
                 /*  SUB     #1,A     ; デクリメント */
```

3.4.7 指標付きアドレッシング

表記 x(rN) は，指標付きアドレッシングを表すために使用される．このモードでは，rN の内容に x を加えて，その加算結果であるアドレスのセルの内容がアクセスされる．オフセット x は負数でもよい．レジスタ自体は変更されない．例えば，A レジスタが 100 を含んでいるならば，MOV 10(A), B 命令は 110 番地（100＋10 だから）のメモリを B レジスタに移す．A レジスタ自体は変更されない．このアドレッシング・モードは，配列の 1 要素をアクセスするのにも使用される．A レジスタが文字配列のベース・アドレスを含んでいるとき，MOV 12(A), B は配列の 13 番目の要素を B レジスタに移す（最初の要素は配列のベース・アドレスからオフセット 0 のところにある）．

配列をアクセスするこの能力を維持するのに，ワード転送はいくぶん複雑になる．初めに A レジスタに 100 が入っていると，MOV.W 12(A), B は A レジスタの内容に 24 をたし，124 と 125 のセルの内容を B レジスタに移す．つまり，ワード転送なので，指標は最初に 2 がかけられる．このかけられた値は希望のアドレスを与えるために，A レジスタの内容に加えられる．それで，この最後の例は，整数配列（文字配列と比べて）の 13 番目の要素をアクセスするだろう．

ワード転送でもうひとつ複雑なことはアライメントである．通常，偶数アドレスからだけ，ワード・サイズのものをアクセスできる．したがって，ワード転送

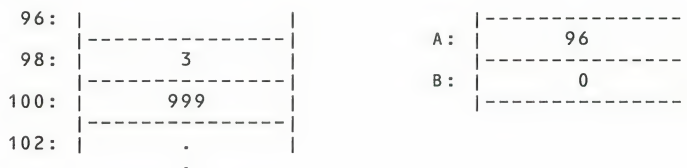



図 3・9　指標付きアドレッシング

をするときは，ベース・アドレスが偶数であるように気をつけなければならない．指標に自動的に2をかけることで，すべての対象アドレスもまた偶数であることを保証している．

C言語では，指標付きアドレッシングは配列をアクセスするのに使われる．次の例を考えてみる．

```
int       array[10];        /* 200―220番地のメモリを使用する00-220 */

array[0] = 1;               /* MOV.W  #200,A  ;  A = 配列のアドレス */
                            /* MOV.W  #1,0(A) ;  A[0] = 1           */
array[6] = 2;               /* MOV.W  #2,6(A) ;  A[6] = 2           */
```

まず配列のベース・アドレスを指し示すように，Aレジスタの初期化から始めることに注意する．C言語では，実際はデータの型として配列をサポートしていない．言いかえれば，配列は本当のオブジェクトではない．むしろ，配列へのポインタが本当のオブジェクトである．例えば，配列をサブルーチンに渡すとき，配列自体を渡すのではない(Pascalでは配列を渡せる)．代わりに，最初の要素のアドレス（最初の要素へのポインタ）が渡される．したがって，C言語では，配

列名は配列の最初の要素のアドレスに評価される．コンパイラが配列名をみたときに，まず行うことは，A レジスタにその配列への隠れたポインタをつくることである．つまり，コンパイラは，A レジスタにその配列のベース・アドレスを移す．すべての配列へのアクセスは，このポインタを経由して行われる．C では

```
array[6] = 2;
```

と

```
char    *xp = array;
*(xp+6) = 2;
```

に違いはない．実際に，多くのコンパイラが，この 2 つに同じ命令コードを生成する．

この例では，計算機にアドレス計算をさせている．整数配列にアクセスして (これはどの要素も 2 バイト使用する)，ワード転送を使うから，MOV.W ♯1,0(A) はメモリ番地 200 と 201 をアクセスする．MOV.W ♯2,6(A) は，番他 212 と 213 を変更する．つまり，(6*int サイズ) が A レジスタにあるアドレスに加えられて (しかし，A レジスタは変更されない)，セルがアクセスされる．配列が構造化されているならば，自分でアドレス計算をしなければならない．

3.5 命 令 セ ッ ト

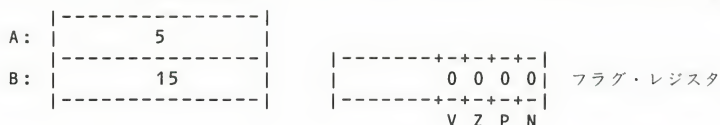
3.5.1 転送 (move) 命令

デモ計算機では，転送命令はひとつだけある．それがどのように使用されるのか，もうすでにいくつかの例でみてきた．

MOV　r1，r2⟷Move　r1　to　r2

3.5.2 算 術 命 令

指定された算術演算を行うことに加えて，すべての算術命令はフラグレジスタのひとつ以上のビットを演算結果に基づいて修正する．特に

結果が 0 ならば，Z フラグが 1 にセットされる．
結果が整数ならば，P フラグが 1 にセットされる．
結果が負数ならば，N フラグが 1 にセットされる．
結果がレジスタをオーバーフローしたら，
　(結果が 16 ビットより大きかったら，)
　　　V フラグが 1 にセットされる．

SUB.W　B,A　　(A レジスタから B レジスタをひき，結果を A レジスタにおく．さらに，結果 (−10) が負数なので N フラグが 1 にセットされる)

ADD.W　#1,B　　(B レジスタに 1 を加える．N フラグは 0 にクリアされ，加算の結果が正数だったので P フラグが 1 にセットされる)

図 3・10　状態フラグ

これらのフラグは，次章で述べる分岐命令で順に使われる．フラグの動作を図 3-10 に示す．算術命令は，2 つのレジスタを変更することに注意すべきである．命令に規定された目的レジスタとフラグレジスタ (演算結果を反映する) である．

ADD r1, r2　　r2 に r1 をたして，結果を r2 におく (r2=r2+r1).

SUB r1,r2　　r2 から r1 をひいて，結果を r2 におく (r2=r2−r1).

AND r1,r2　　ビットごとに r1 と r2 を AND して，結果を r2 におく．

OR r1,r2　　ビットごとに r1 と r2 を OR して，結果を r2 におく．

NOT rN　　ビットごとに rN を反転 (1 の補数に) する．

SHR x,rN　　rN を x ビット右にシフトする ($0<x<16$)．最上位のビットは 2 重化される (つまり，最上位のビットが 1 ならば，1 が残される．そうでなかったら，0 が残される)．この 2 重化はシフトされる数の符号を維持する．

SHL x,rN　　rN を x ビット左にシフトする ($0<x<16$)．もっとも右のビットに 0 が入れられる．

例 ADD 2000,(A) は A レジスタに含まれているアドレスのセルの内容を，アドレス 2000 にあるセルの内容に加える．結果は，A レジスタに含まれているアドレスのメモリに格納される．SUB #2,C は，C レジスタの内容から 2 をひく．ADD C,#2 は誤りである．ひとつの命令には代入演算子が隠れていることに注意しなさい．たとえば，x+=y は直接 ADD.W y,x に変換される．

3.5.3 ジャンプ命令（分岐命令）

ジャンプ命令は，次の番地にある命令以外のところを指し示すように，プログラム・カウンタ（実行される命令のアドレスを含んでいる）を修正して行う．JMP は無条件ジャンプ命令である．単に指示されたアドレスにある命令を実行し始める．指示されたアドレスに命令がなければ，予想できないことが起きる．他のジャンプ命令（JP，JGE 等）は，対応するフラグの状態が真のときだけ実行される．

JMP rN	PC=rN
JP rN	P フラグが 1 にセットされているときだけ，PC=rN
JGE rN	P か Z フラグが 1 にセットされているときだけ，PC=rN
JZ rN	Z フラグが 1 にセットされているときだけ，PC=rN
JN rN	N フラグが 1 にセットされているときだけ，PC=rN
JLE rN	N か Z フラグが 1 にセットされているときだけ，PC=rN
JV rN	V フラグが 1 にセットされているときだけ，PC=rN
JNV rN	V フラグが 1 にセットされていないときだけ，PC=rN

他の命令も，次に示す例のように，ジャンプを行うことができる．

```
ADD  #128,PC       PC = PC + 128.
SUB  #4,PC         PC = PC - 4.
ADD  (C),PC        PC = PC + Cレジスタ中のアドレ
                         スにあるセルの内容.
```

ジャンプ命令は **goto** 分岐，**if/else** 文，**switch** 文，**while** や **for** のような繰返し文を処理するために使用される．それらの文は，後でもっと深く調べる．サブルーチンの呼出しは，別のメカニズムでも行われるので，次のセクションと次章で調べる．

3.5.4 サブルーチン・コールとスタック

2つのジャンプ命令は，独自のセクションを設けてもよいほど重要である．それはJSR（サブルーチンへのジャンプ）と，RET（サブルーチンからのリターン）命令である．サブルーチンは，プログラム内のどこからでも制御を移されることのできる，これだけで独立したプログラムである．サブルーチンは，仕事をしてから，サブルーチンを呼び出した命令の次の命令に制御をかえす．どこからジャンプしたのか知る必要があるので，通常のジャンプ命令ではサブルーチンに行けない．この，もどってくる場所をリターン・アドレスという．リターン・アドレスは，サブルーチンがまたサブルーチンを呼ぶこともあるので，固定したメモリ番地に格納することはできない．なぜならその2回目の呼出しが，最初の呼出しで書かれたリターン・アドレスに上書きしてしまうからである．この問題の解決法は，スタックと呼ばれるデータ構造である．一般の物にたとえると，カフェテリアでのお盆の山積みである．最後に上におかれた盆が最初に取られる．最後にスタックに入れられたデータが最初に取り出される．スタックにデータを置くことをプッシュ（push），データを取り出すことをポップ（pop）という．SP，またはスタック・ポインタ，レジスタはスタックの管理に使われる．SP レジスタはいつ

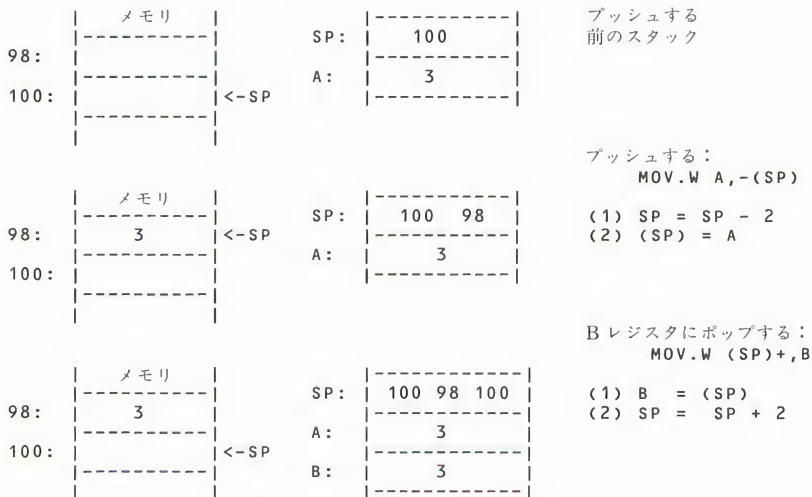



図 3・11　プッシュ操作とポップ操作

も，スタックにもっとも新しくオブジェクトがプッシュされたメモリのアドレスを保持する．MOV.W rN,－(SP) 命令でスタックに何かをプッシュし，MOV.W (SP)＋,rN 命令でポップすることができる．この処理がどのように行われるか詳しくみてみよう（図3·11参照）．

スタック・ポインタは，現在スタックの1番上にあるメモリ・セルのアドレスを保持している．プッシュ操作は次の2つのことを行う．まず，スタック・ポインタを減少させる．そして，スタック・ポインタ（減少後）に現在あるアドレスのメモリにオブジェクトをプッシュする（この場合Aレジスタの内容）．Aレジスタのプッシュは C 言語では＊－－SP＝A のように表す．

ポップ動作は，プッシュ動作の逆である．ここでは MOV.W (SP)＋,B 命令を使う．まず，スタックの1番上にあるオブジェクトを（このオブジェクトを含むセルのアドレスがスタック・ポインタに含まれている），目的の場所（B レジスタ）に移動する．それからスタック・ポインタを減少させる．ポップに対する C 言語の文は，B＝＊SP＋＋ である．スタックから取り除かれたときポップされたオブジェクトは，実際はスタック上で消されないけれども，ポップ後は絶対に使うべきではない．割込み処理ルーチンとコンパイラの実行時ライブラリによって起こる問題がある．どちらもスタックを使用するのである．これらのルーチンの実行に気がつかないために，貴重なスタックの値に上書きされてしまいやすい．オブジェクトがポップされたとき，他のみえないルーチンによってスタック上でそのオブジェクトに上書きされていないか確かめる方法はない．

プッシュとポップは，一般的な動作であるので，このアセンブラでも専用のニーモニックをサポートする．

```
PUSH   rN  は  MOV.W   rN,-(SP)   に変換される.
POP    rN  は  MOV.W   (SP)+,rN   に変換される.
```

スタックは，16ビット幅のデータ構造である．同じスタック上に，16ビットと8ビットのオブジェクトをおくと管理が難しくなる．それで，ワードサイズのオブジェクト用の命令と異なり，バイトサイズのオブジェクト専用の命令はない．

スタックについて注意することがいくつかある．

（1） 前置きデクリメントを使うと，スタックポインタはいつも最後にプッシュされたオブジェクトを指し示すことが保証される．このように，スタックの1番上のオブジェクトは，スタックポインタを通して間接的にアクセス

される.

（2） 連続してプッシュしても，オブジェクトは別のメモリ番地におかれる．なぜなら，プッシュの度にSPが減少されるからである．その結果，後のプッシュで，前にプッシュされたものに上書きすることはない．

（3） スタックはメモリの下方に（下位番地の方向に）伸びる．

ルール(3)は，特にマイクロコンピュータでは，いくつかの結果をもたらす．すでに，他のものを含んでいるメモリの領域を，スタックが浸食しているかチエックしない．スタックが大きくなりすぎた状態をスタック・オーバーフローと呼び，Cプログラムの特にやっかいなバグの原因になる．多くのCコンパイラは図3·12のようにメモリを使用する．命令コードは，メモリのもっとも低いところにおかれる．静的データはその次におかれ，スタックは上位のメモリにおかれる．スタックは下方に伸びる．それで，スタックポインタはスタック領域のもっとも上位の番地を指すように初期化される．IBM PCのような計算機では，メモリの保護機能をサポートしていない．スタックが大きくなりすぎて，データ領域に上書きしたり，重傷の場合は，命令コード領域に上書きしてしまうこともありうる．データ領域に上書きしたら，広域変数はその値をかえ，意味がなくなってしまう．コード領域に上書きしたら，CPUは正当な命令だと思って，ゴミになったコードを実行し始めてしまう．次章で，スタックのオーバーフローの原因をいくつかみてみる．

たいていのメインフレームでは，スタック・オーバーフローの状態が進展することを許さない．オペレーティング・システムは，まずプログラムを終了させる．同じことが危険なポインタにも適用される．プログラムが，命令コード領域に上書きしようとしたら，オペレーティング・システムはプログラムを終了させる．

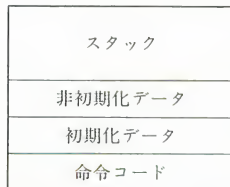



図 3·12 Cプログラムのメモリ使用

UNIX では，このような異常な動作が起こると，"segmetation violation, core dumped"（セグメンテーション違反）のエラー・メッセージを生成し，違反を起こしたプログラムを終了させる．スタック・オーバーフローと危険なポインタは，マイクロコンピュータでは現実の問題である．

```
        |-----------|
10:     |  JSR 20   |<-PC
        |-----------|
12:     |           |

        |-----------|
20      |    RET    |
        |-----------|
                                |-------------|
98:     |           |      SP:  |     100     |
        |-----------|           |-------------|
100:    |           |<-SP  PC:  |      10     |
        |-----------|           |-------------|
```

JSR をする前の状態：

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

        |-----------|
10:     |  JSR 20   |
        |-----------|
12:     |           |

        |-----------|
20      |    RET    |<-PC
        |-----------|

        |-----------|           |-------------|
98:     |    12     |<-SP  SP:  |  100   98   |
        |-----------|           |-------------|
100:    |           |      PC:  |   10   20   |
        |-----------|           |-------------|
```

JSR をした後の状態：
JSR に続き命令のアドレスはスタック（98 番地）にプッシュされ，PC レジスタは JSR のひき数と置き換えられる．
今は 20 番地の命令を実行している．

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

        |-----------|
10:     |  JSR 20   |
        |-----------|
12:     |           |<-PC
        |-----------|

        |-----------|
20      |    RET    |
        |-----------|
                                |-------------|
98:     |    12     |      SP:  | 100 98 100  |
        |-----------|           |-------------|
100:    |           |<-SP  PC:  |  10  20  12 |
        |-----------|           |-------------|
```

RET 命令実行後の状態：
リターン・アドレスはスタックから PC レジスタのポップされた．
今は 12 番地を実行している．

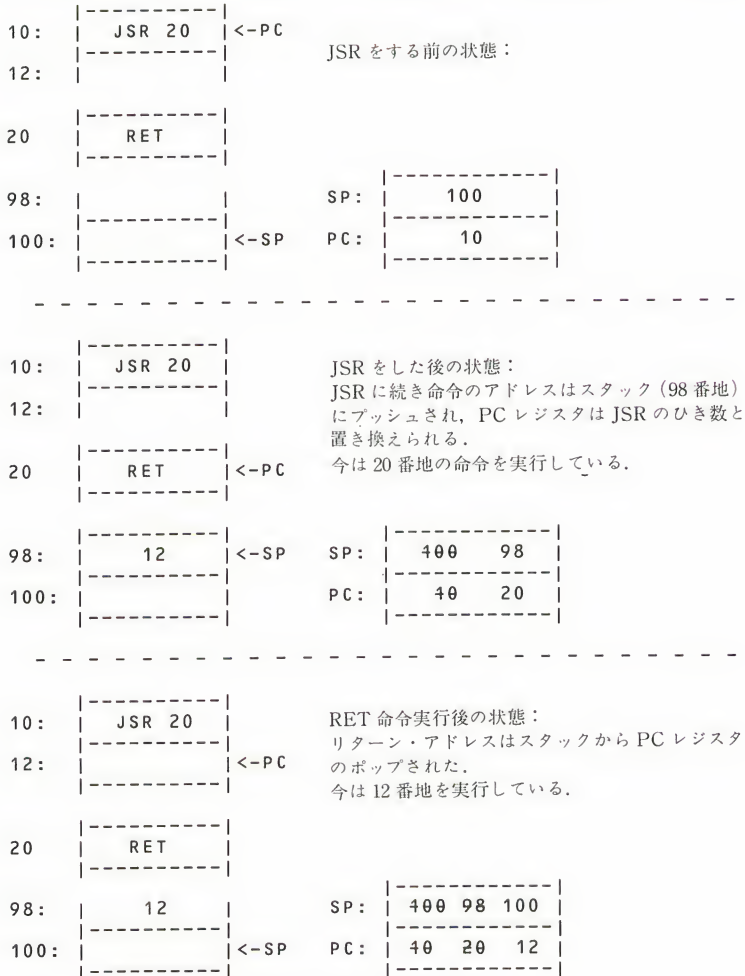



図 3・13　JSR と RET 命令

さて，もとの話題，JSR 命令と RET 命令にもどることにしよう．JSR 命令は2つのことをする．JSR 命令に続く命令のアドレス（リターン・アドレス）をスタックにおき，そして，命令の引数であるアドレスに制御を移す．RET 命令は，スタックの1番上のものをポップして，PC に入れ，もとの JSR に続く命令に制御をもどす．この過程が図 3·13 に示されている．

JSR rN　　アドレス rN にあるサブルーチンに進む．

　(1) SP=SP−2

　(2) (SP)=PS+2

　(3) PC=rN

RET　　サブルーチンからもどる．

　(1) PC=(SP)

　(2) SP=SP+2

rN は，間接アドレスでもよい．JSR (A) は，A レジスタにあるアドレスのサブルーチンにジャンプする．この場合は，A レジスタがサブルーチンへのポインタであり，C 言語では，正規の使用法である．

3.6 ラ　ベ　ル

第1カラムから書かれた1字から8字までの文字列に，ひとつのコロンが続くものをラベルと定義する（命令名やレジスタ名 JMP，ADD，A，B，SP，コロン，などはラベルとして使わない）．アセンブラがラベル名（コロンは除く）をみつけると，ラベルをそのラベルに続く最初の命令のアドレスと置き換える．このように，メモリ中の実際の位置を求める必要はない．JSR の例では，次のように書ける．

```
        JSR     subr

subr:   RET
```

JSR 命令中の subr は，自動的に RET 命令のアドレスと置き換えられる．

アセンブラは，シンボル・テーブルを使ってラベル名を維持する．アセンブラがラベル名を発見するたびに，ラベル名とそれに関係するアドレスをシンボル・テーブルに登録する．ラベルが使用されるたびに，シンボル・テーブル内にその名前を探して，それに関連するアドレスでラベルを置き換える．たいていアセンブラは前方参照（さきほどの例のように，ラベルが定義される前に使われるよ

うな場合）を処理するために2回プログラムをみる．C言語のシンボル・テーブルは，もっと複雑であるけれども，Cコンパイラもシンボル・テーブルを使用する．

3.7 コ メ ン ト

1行の中で，セミコロンの後にある文字は，すべてアセンブラから無視される．コメントは，プログラムが何をしているのかを，別のドキュメントに書かないで，同じプログラム中で説明するのに使う．

3.8 疑似命令と静的初期化

ある予約語は，アセンブラに，コード生成以外のことをするように伝える役目を行う．これらの予約語は，疑似命令と呼ばれる．なぜなら，それらは命令のような形をしているけれども，本当の命令ではないからである．疑似命令

```
label:  DB      N
```

は，アセンブラに1バイトのメモリを確保させ，数Nに初期化し，そのバイトのアドレスに対するラベルをつくる（DBはdefine byteを表す）．つまり，命令を出力するのではなく，数Nをメモリに入れるのである．

```
label:  DW      N
```

は，1バイトではなく，1ワード確保する．

C言語には，staticのオブジェクトの初期化文は，次のようにDBかDWを生成する．

```
static int romeo;     /*  romeo:  DW 0                  */
static int juliet;    /*  juliet: DW 0                  */
romeo  = 5;           /*          MOV.W  #5,romeo       */
juliet = romeo;       /*          MOV.W  romeo,juliet   */
```

この初期化されたメモリ領域は，命令コードといっしょにディスク上に保存される．プログラムが，オペレーティング・システムによってメモリ上にロードされるとき（つまり，実行されるとき），初期化されているデータは正しい値を持ってメモリ中の正しい位置に読み込まれる．この初期化を行う命令コードは生成されていない．データは，正しい初期値を持って，ただディスクから読み込まれるだけである．これは，サブルーチンが最初に実行されるときだけ，そのサブルーチンにローカルな静的変数が初期値を持っている理由を説明している．この変数は，いったん変更されると新しい値を保持する．この動作は，明確に初期化され

ていない静的変数が，0に初期化される理由も説明している．プログラマは，セルを何かに初期化しなければならない．

3.9 練　　習

3-1　2章で述べたアルゴリズムを使用して，AレジスタとBレジスタに保持されている8ビットの数をかけ合わせて，結果をCレジスタにおくプログラムを書きなさい．

3-2　文字列を移動するアセンブリ言語のプログラムを書きなさい．Aレジスタは文字列の最初の文字のアドレスを含んでいる．Bレジスタは文字列のバイト数を含んでいる．Cレジスタは，文字列が移動される領域のアドレスを含んでいる．もとの文字列と移動後の文字列は，重なっても構わない．この場合，もとの文字列は修正されるが，移動後の文字列がもとの無修正の文字列を含んでいる．

3-3　次のプログラムで，すべてのレジスタ（SPとPCは含まれるが，FPは含まない）の内容を表示しなさい．

(a)　JSR命令の直前．

(b)　ラベルsubrに続くMOV命令が実行される直前．

(c)　RET命令が実行される直前．

(d)　HALT命令が実行される直前．

すべての命令は2バイトで，最初の命令はメモリ100番地にあるとしている．

```
start:  MOV     #100,SP
        MOV     #0,A
        MOV     A,B
        ADD     #10,B
        MOV     B,C
        SHL     3,C
        MOV     C,D
        ADD     B,D
        NOT     D
        JSR     subr
        HALT                    ; プロセッサを止める
subr:
        MOV     A,-(SP)
        MOV     B,-(SP)
        MOV     C,-(SP)
        MOV     D,-(SP)
        MOV     (SP)+,A
        MOV     (SP)+,B
        MOV     (SP)+,C
        MOV     (SP)+,D
        RET
```

4　コード生成とサブルーチン結合

本章では，前章で学んだアセンブリ言語の使用をすすめる．C 言語のサブルーチンの呼出しとパラメータの受渡しを，サブルーチンの中からどのように変数をアクセスするのかをみながら，より詳細に調べる．そしてフロー制御文を理解するために簡単なコード生成について述べる．

引数の受渡しの規則をよく知っていれば，再帰呼出しや不定数の引数を持ったサブルーチンを書くのに役立つ．変数が，内部でどのように配置されているのか理解していることはとても重要なことであり，デバッグでたいへん貴重な手助けになる．実際，バグのなかには，それを知らなくてはみつけることがほとんどできないものもある．

4.1　サブルーチンの呼出し

このセクションでは，制御がサブルーチンに渡されるとき生成されるコードについて述べる．コンパイラに生成された実際のコードはもっと複雑だが，ここでの例は重要ないくつかの概念を説明するのに役立つだろう．

最初に，いくらか基礎知識を必要とする．本章では実行時（run-time）とコンパイル時（compile-time）ということばを使用する．コンパイラが動いて，コード生成が行われている間をコンパイル時という．コンパイル時にすることが多いほど，プログラムが実際に実行している間（実行時）にしなければならないことは少なくなる．例えば，定数式（数だけで，変数を含まない式）は，評価するための情報をコンパイラがすべて持っているから，コンパイル時に評価されるべきである．このように，命令コードを用いて式を評価する必要はないから，プログラ

ムは，コンパイルをするためにわずかな時間を必要とするだけで，よりはやく動作する．例えば，式x ＝（1024＊3）は，1024に3をかけるために必要なかなりの量のコードを生成しないで，ひとつの命令MOV.W ＃3072,xを生成する．

前の章では扱わなかったレジスタについても述べる．フレーム・ポインタ・レジスタ（FP）は，スタック・ポインタ（SP）と同様に，スタック上のメモリ・アドレスを保持している．しかし，SPはセル単位の参照が変化する（プッシュとポップのたびに更新される）が，FPはスタック上の大きな領域への参照が固定している．フレーム・ポインタは，サブルーチンの変数を維持するために使われる．すべてのローカル変数は，サブルーチンに入ったとき（実行時）つくられたひと固まりのブロック内のスタックに維持される．サブルーチンの引数も変数と同じメモリ・ブロック内のスタックに保持される．このメモリ・ブロックは，サブルーチンのスタック・フレームと呼ばれる．そして，フレーム・ポインタは，スタック・フレームとして構成されるさまざまなメモリ領域を参照するのに使用される．

スタック・フレームは，サブルーチンに入るときつくられる．そして，サブルーチンからもどるとき（プッシュとポップに似たやり方で）消去される．したがって，メモリ（スタック）の同じ領域を，さまざまなサブルーチンが実行するときにいつも再利用する．つまり，いくつかのサブルーチンがそれぞれのローカル変数用に，スタックの同じ部分を使用する．

しかし，すべての変数がスタック上にあるわけではないので複雑である．特に，レジスタ変数はレジスタにおかれる．広域変数（サブルーチンの本体の外で宣言されている）は，はっきり予約語staticで宣言されたローカル変数のように，固定されたメモリ番地にある．スタック上にある変数は，自動変数（automatic variables auto変数）と呼ばれる．

図4·1に，短いCプログラムとそれに対応するアセンブリ言語を示す．一見するとこのプログラムは複雑そうだが，心配する必要はない．本章の残りでそれを理解すればよい．

プログラム自体を深くみてみる前に，図4·1の行Mにあるfoo( )への呼出しを例にして，プログラムが行う動作をみてみよう．まず，foo( )用のスタック・フレームがつくられる．スタック・フレームの半分は，呼ぶ方のサブルーチン（この場合はmain( )）によってつくられ，後の半分はfoo( )自体がつくる．

```
A:              int  p1 = 1, p2 = 2, p3 = 3;
B:
C:              foo( p1, p2, p3 )
D:              int     p1, p2, p3;
E:              {
F:                      int     v1, v2, v3;
G:
H:                      return 0;
I:              }
J:
K:              main( )
L:              {
M:                      foo( p1, p2, p3 );
N:              }

----------------------------------------------------------------
1:   _p1:       DW 1              ; static グローバル変数p1, p2, p3 にメモリを確保
2:   _p2:       DW 2              ;     する.
3:   _p3:       DW 3
4:
5:   _foo:                        ; スタック・フレームの設定.
6:              PUSH  FP          ;   フレーム・ポインタの待避
7:              MOV.W   SP,FP     ;   新しいフレームの設定
8:              SUB.W   #6,SP     ;   ローカル変数のメモリを確保
9:
10:             MOV.W   #0,D      ; 返り値の設定
11:                               ; 前のフレーム・ポインタの回復
12:             MOV.W   FP,SP     ;   ローカル変数の消去
13:             POP   FP          ;   前のフレーム・ポインタの回復
14:             RET
15:
16:  _main:                       ; スタック・フレームの設定
17:             PUSH FP           ;   フレーム・ポインタの待避
18:             MOV.W   SP,FP     ;   新しいスタック・フレームの設定
19:             SUB.W   #0,SP     ;   ローカル変数はない
20:
21:                               ; サブルーチン "foo(p1, p2, p3):" の呼出し
22:             PUSH _p3          ;   最右のパラメータ (p3) をプッシュ
23:             PUSH _p2          ;   中央のパラメータ (p2) をプッシュ
24:             PUSH _p1          ;   最左のパラメータ (p1) をプッシュ
25:             JSR     _foo      ;   サブルーチンを呼びだす
26:             ADD.W   #6,SP     ;   スタック上の引数を消去する
27:
28:                               ; 前のスタック・フレームの回復
29:             MOV.W   FP,SP     ;   ローカル変数の消去
30:             POP   FP          ;   前のフレーム・ポインタの回復
31:             RET
```

図 4・1 C プログラムとアセンブリ言語

main( ) は，スタックにプッシュして引数を foo( ) に渡す．このプッシュされた引数は，スタック・フレームの最初の部分にある．引数は右から順にプッシュされる（p3 が最初にプッシュされる）．C 言語では，すべての変数の受渡し（配列は除く）は値で行われることに注意するべきである．つまり，変数 p3 自体ではなく，p3 の値が foo( ) に渡される．別の見方をすると，main( ) が p3 の内容をスタックにプッシュして，p3 のコピーを行う．このやり方だと，foo( ) が p3 に対応する引数を変更しても，実際にはもとの変数自体は変更されずに，変数

のコピーが変更される．変数がプッシュされた後のスタックを図4·2に示す．

もとのp1，p2，p3の値（内容）はスタック上にプッシュされる．そのアドレスは簡単に示されている．まだmain( )に制御はある

図 4·2　スタックにプッシュされた foo( ) へのパラメータ

次に，サブルーチン foo( ) が呼びだされる．この呼出しはスタックにリターン・アドレスをおく（よくわからなかったら，前章のJSRとRET命令を復習するとよい）．この新しいスタックが，図4·3に示されている．

```
          |                 |
          |-----------------|
  998:    | リターン・アドレス |   <-SP
          |-----------------|
 1000:    |        1        |
          |-----------------|
 1002:    |        2        |
          |-----------------|
 1004:    |        3        |
          |-----------------|
 1006:    |                 |
```

サブルーチン foo( ) に入っているが，まだ命令コードはまったく実行していない．

図 4·3　foo( ) に入った直後のスタック

foo( ) が最初に行うことは，現在のフレーム・ポインタをプッシュして，そ

れを保存することである．それから現在のスタック・ポインタをフレーム・ポインタにコピーする．図4·4はコピーした後のスタックを示す．

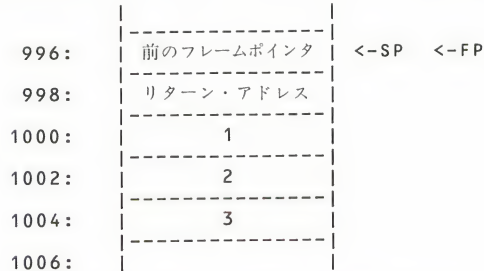



図 4·4 フレーム・ポインタを待避した後のスタック

次に，foo( ) は自分のローカル変数にメモリを割り当てる．このメモリはスタック・ポインタから適当な大きさの定数をひいてスタックから得られる．図4·5にメモリを割り当てた後のスタックを示す．

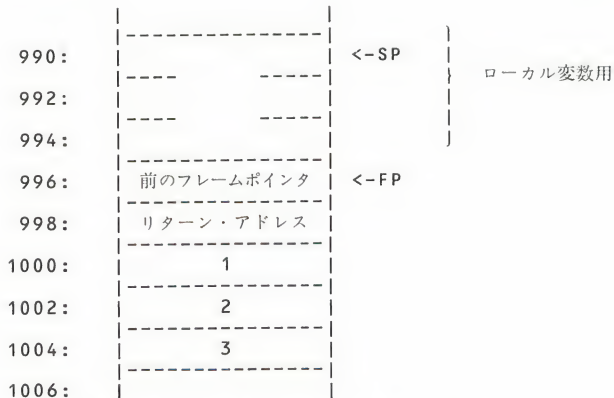



図 4·5 スタック・フレームの残りがつくられる

この時点で，スタック・フレームが完成する．しばらくの間，この構成がどのように使用されるかをみることにする．そして，サブルーチンからもどるとき何が起こるかみてみよう．foo( ) は自分でつくったスタック・フレームのすべての部分を削除しなければならない．ローカル変数は，フレーム・ポインタの現

在の値をスタック・ポインタにコピーし，スタック・ポインタがフレーム・ポインタと同じ場所をさすようにして削除される（図4·6を参照）．次に，フレーム・ポインタには，図4·7に示されるように，スタックから前の値がポップされて再格納される．それからfoo( )はRET命令を実行し，main( )にもどって，スタックからリターン・アドレスをポップする．スタックは現在，foo( )が呼ばれる前と同じようにみえる（図4·8をみよ）．

ここで，main( )は引数に関連したスタック変数を捨てる．呼びだされたサ

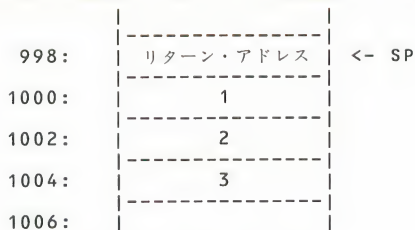
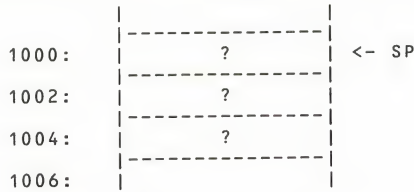
図 4·6　ローカル変数の削除

まだサブルーチン foo( ) にいる

図 4·7　前のフレーム・ポインタをポップで回復する

main( ) にもどってきたところ
図 4・8 foo( ) からリターンした後のスタック

ブルーチンに渡した引数はmain( ) では使用されないから，スタックからポップしてもとの変数にわざわざもどすことはない．したがって，サブルーチンが他のサブルーチンに使用された変数を直接変更する方法はない（直接ではなく，ポインタを使用して間接的に変更することはできる）．この変数はスタック・ポインタに数をたして削除される．つまり，図4・8に示すようにスタック・ポインタに6を加えて，アドレス1006をさすようにし，その結果引数が削除される（引数は3ワード・サイズである．3*2バイト ＝ 6バイト）．

少しもとにもどって，このすべてのプッシュとポップを行う命令コードをみてみよう．命令コードはコンパイル時に生成され，コンパイラは始めから終わりまで1度その部分を通過するだけである．いいかえれば，コードは入力ラインがあると生成される．サブルーチンが呼ばれる順番（実行時）は最終的なプログラムに現れる順序に関係ない．main( ) が最初に実行されるサブルーチンであるからといって，最初に生成されるコードであるとか，目的モジュールの始めにおかれる命令コードであることを意味しない．

図4・1の下の部分をみると，コンパイラが名前を変更している（先頭に下線を加えている）ことに気づくだろう．コンパイラはしばしば，サブルーチンを呼んで複雑な動作（スイッチにある何かの捜索や浮動小数の乗算）を行う．これらのサブルーチンは，コンパイラのメーカーから実行時ライブラリとして供給されている．コンパイラは，それらのサブルーチンは利用できると仮定して，それらのサブルーチンの呼出しを生成する．すでに定義されているこれらのサブルーチンの名前には，下線で始まるものはない．同様に，実行時ライブラリで使われている外部データには，名前が下線で始まるものはない．コンパイラは，実行時ライブラリにすでにあるものと決してかちあわないように，読者の変数に下線を加え

る．この付加された文字はリンカから出力されるリンク・マップでみることができる（コンパイラの中には下線以外の文字を使用するものもある．また，先頭ではなく他の位置に文字を付加するコンパイラもある）．さらに，Cプログラムから呼びだされるアセンブリ言語のルーチンを書くときには，アセンブリ言語にこの付加された文字を加えなければならない（Cプログラムには必要ない）．

次にstatic変数のメモリ割当てをみてみよう．図4·1の1～3行目はCのソース・プログラムのA行をみてコンパイラが生成した．これらの変数は，広域変数である（これらはサブルーチンの外で宣言されている）から，DW命令を使って固定したメモリ位置に割り当てる．p3がDW命令の一部で3に初期化されていることに注意しなさい．前章でも述べたように，static変数の初期化には命令コードは使われない．C言語の2つの基本的な記憶クラス，staticとautoは，根本的に異なるやり方で扱われる．static変数はすべて固定したメモリ位置にある．それで，DW命令でメモリを確保する．後でみるように，auto変数は違ったやり方でつくられる．

さきに進んで，サブルーチンの呼出しをみてみよう．図4·1のM行にあるサブルーチンfoo( )への呼出しがアセンブリ言語のプログラムの22～26行目を生成する．引数はスタックでサブルーチンに渡されることを思いだしなさい．スタックに引数をプッシュすることと，呼びだされたサブルーチンからかえった後でスタックを消去するのは，呼びだしたサブルーチンの責任である（この手続きは，サブルーチンに不定数の引数を持たせることができる理由のひとつである．呼びだす方のルーチンが，スタックを管理している限り，不定数の引数を持ったサブルーチンは，前もって引数の数を知る必要はない）．引数は逆の順番（右から左に，p3は最初に）プッシュされる．3つの引数は，宣言されたグローバル変数である．したがって，固定のメモリ位置にあり，そして，22～23行目のプッシュ命令で，直接そのアドレスを参照することができる．サブルーチンは，実際は25行目で呼びだされる．JSR命令も，リターン・アドレスをスタックにプッシュすることを思いだしなさい．

foo( )からもどってきて（26行），スタックは呼びだされる前の状態になる．コンパイラは，ADD.W　#6,SP命令でスタックから引数を消去する．引数は捨てられる．それで引数をポップする代わりに，スタック・ポインタに定数を加える（これは，実際には何も移動しないがポップを繰りかえすことと等しい）．

状態は，行8のSUB.W（ローカル変数のためにメモリをつくるために使われる命令）に似ている．つまり，3つのポップ命令は，スタック・ポインタに6を加える（ワード移動であるから）．そして，スタックから取りだしたものをどこかへおく．スタックに何も残しておきたくないから，MOV 命令はしないでただ直接たし算をする．

さて，サブルーチン自体をみてみよう．foo( ) に入るとき，スタック・フレームの一部は引数をプッシュしてつくられる．フレームの残りをつくるのは呼びだされたサブルーチンの責任である．サブルーチン foo( ) はソース・プログラムではC行で，アセンブリ言語では行5から始まる．C，D行はシンボル・テーブル（後で詳しく説明）に影響する．しかし，命令コードは何も生成されない．コンパイラは '{'(E行) にあうと，6行の PUSH FP で前のフレーム・ポインタを保存する．プッシュの後，現在のスタック・フレーム（いまつくっている）に対するフレーム・ポインタが，行7の MOV.W SP, FP で初期化される．現在のスタック・ポインタの内容をフレーム・ポインタにコピーしている．

コンパイラは，行F でローカル変数をみつける．行8にある SUB.W で foo( ) のローカル変数のメモリを確保する．SP を6だけ減少させ（これもワード移動である．ワードサイズの3倍は6）0を3つスタックにおくためには3 PUSH #0 命令を使うこともできるだろう．しかし，ローカルの auto 変数は，サブルーチンに入るとき意味のない値であるから，3つプッシュを使うより，SUB 命令でスタック・ポインタを直接操作したほうが効率がよい．SUB.W は，スタック上にローカル変数用のメモリをつくる．スタック・ポインタを安全な場所（ローカル変数の下方）に移動するだけである．

サブルーチンが呼ばれるごとに，スタック上にローカル変数用のメモリをつくり，リターンする前にそれらの変数をクリアする．このように，メモリの同じ領域（スタック）がさまざまなサブルーチンのローカル変数に繰りかえし使用される．

foo( ) の本体は，行H にあるたったひとつのリターン文からなる．一般的に，値は呼出しをしたサブルーチンにレジスタでかえされる．foo( ) は，行10の MOV.W #0, D 命令で呼出しをしたルーチンに0をかえす．リターン・レジスタには，必ず何かがおかれている．それは，return 文がサブルーチンに書かれているかどうかには関係ない．呼びだされた関数が値をかえさないときには，リタ

ーン・レジスタに保持されているものは意味がない．かえされた値が有効なものであるか，呼出しをしたルーチンが知る手立てはないから，ひとつのサブルーチンに複数のリターン文があるときは危険である．それらのリターン文のひとつが有効な値をかえさない（引数を持たないリターン文を実行する）のならば，意味のない値が時にはかえされることになる．同様に，はっきりリターン文が書かれていないならば，そのサブルーチンの '}' が実行されるとき暗黙にリターン文が実行され，その暗黙のリターン文は意味のない値をかえす．C では，すべてのサブルーチンが値をかえすことに注意する必要がある．呼びだしたサブルーチンが返り値を使用しないならば，その時は返り値が実際意味のないものであってもたいしたことではない．

実際にリターンを行うのに必要な命令コードは，図 4·1 の 12～14 行目につくられている．サブルーチンは，もとの状態（サブルーチンに入るときの）にもどされたスタック・ポインタとフレーム・ポインタを持って，呼出しをしたサブルーチンにかえらなければならない．これが 12～13 行目で行われている．フレーム・ポインタ（FP レジスタ）の現在の内容をスタック・ポインタ（SP）に移動することでスタックがもとにもどされ，スタック・ポインタはグローバル変数がつくられる直前の状態になる．次に，フレーム・ポインタが POP(FP) でもとの状態にもどされる．最後に，行 14 でリターン・アドレスを PC にポップして，呼出しをしたルーチンにもどる．

4.2 スタック・フレーム

図 4·1 の行 6 に達したときの計算機の状態を図 4·9 に示す（書かれているアドレスは短縮形である）．

繰りかえすことになるが，サブルーチンに使用されるスタックの一部がそのサブルーチンのスタック・フレームである．そして，フレーム・ポインタ・レジスタ（FP）は固定された場所を参照し，スタック・フレーム内の変数をアクセスするために使用される．フレーム・ポインタは，スタック・ポインタと同様に，スタックのある場所のアドレスを保持し，引数はいつもフレーム・ポインタから計算されたオフセットにある．例えば，サブルーチン呼出しの最左にあるパラメータ（この例では p1）は，フレーム・ポインタから +4 のオフセットにある．（図 4·9 を見直しなさい．前のフレーム・ポインタが，現在の FP から +0 の

```
              レジスタ                                 スタティック・メモリ

                                                  アドレス：  |              |
                                                              |--------------|
             |---------------|            _p1     500:  |      1       |
        SP:  |      102      |                           |--------------|
             |---------------|            _p2     502:  |      2       |
        FP:  |      108      |                           |--------------|
             |---------------|            _p3     504:  |      3       |
                                                          |--------------|
                                          _foo    506:  | fooのコード  |
                                                          |   の始まり   |
                                                          |              |
  スタック：

  アドレス：      |               |  フレーム・ポインタ
                  |               |  からのオフセット
   100:           |               |  (バイト)：
                  |---------------|
   102:  SP->     |      v3       |  -6   |                     |
                  |---------------|       |                     |
   104:           |      v2       |  -4   } 3  ローカル変数     |
                  |---------------|       |                     |
   106:           |      v1       |  -2   |                     |
                  |---------------|                             |
   108:  FP->     |    古いFP     |   0                         } foo( )のスタック
                  |---------------|                             | フレーム
   110:           | リターン・アドレス | +2                     |
                  |---------------|                             |
   112:           |      p1       |  +4   |                     |
                  |---------------|       |                     |
   114:           |      p2       |  +6   } 3  引数             |
                  |---------------|       |                     |
   116:           |      p3       |  +8   |                     |
                  |===============|          ---------------
   118:           |    古いFP     |                             |
                  |---------------|   ローカル変数はない        |
   120:           | リターン・アドレス |                        |
                  |---------------|       |                     } main( )のスタック
   122:           |     argv      |       |                     | フレーム
                  |---------------|       } 2  引数             |
   124:           |     argc      |       |                     |
                  |===============|       |                     |
   126:           |               |
```

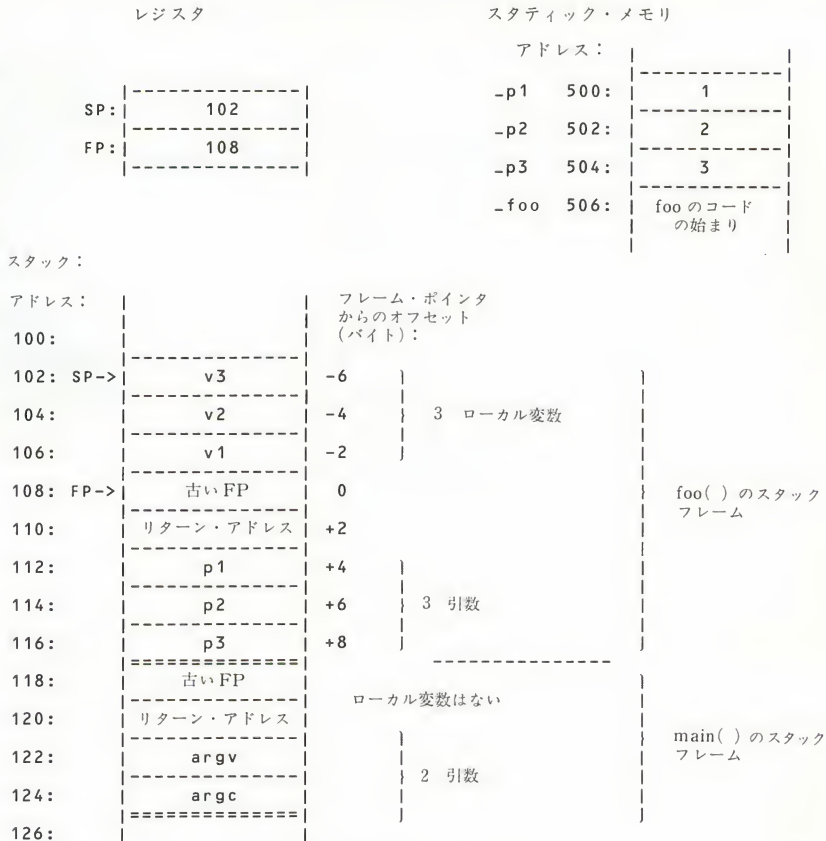



図 4・9 スタック・フレーム

オフセットにある)．パラメータは，このオフセットによってアクセスされる．例えば，パラメータp1は

```
MOV.W   4(FP),rN
```

でfoo( )からフェッチされることができる．p3は

```
MOV.W   8(FP),rN
```

命令でフェッチされる．ローカル変数も，フレーム・ポインタからのオフセットを使って参照される．このときは負の方向である．ローカル変数p3は

```
MOV.W   -6(FP),rN
```

でfoo( ) の中から参照される．再び図4·9を参考にして，v1はFPから −2のオフセットに，v2は −4のオフセットに，v3は −6にオフセットにある．さまざまな変数や，引数のフレーム・ポインタからのオフセットは，それぞれの型によって異なるので注意が必要である．つまりlong intはただのintよりもスタック上で大きな変位をとる．引数の大きさは，2か所で計算されることに注意するべきである．1回は呼出しをするルーチンで，もう1回は呼びだされるルーチンで行われる．呼出しをしたルーチンで宣言されたパラメータの型（int，long，float等）は，コンパイラにスタックにプッシュするのに何バイト必要かを示す．呼びだされるルーチンのパラメータ並びの中の対応する引数に宣言された型が，コンパイラに特定の引数がフレーム・ポインタから何オフセットに発見されるかを示す．コンパイラは，これら2つの宣言が同じであるかどうか判断しない．それで，呼びだす方のサブルーチンが，longサイズのものをスタックにプッシュし，そして呼びだされるルーチンが，intサイズのものを要求しているならば，すべ

```
foo( ) が期待するスタック：                     グローバル変数p1が4バイトのlong
                                                integerに宣言されたならば，つくら
                                                れるスタック：

               フレーム・ポインタ                              フレーム・ポインタ
アドレス：     からのオフセット：               アドレス：     からのオフセット：

                                                          |---------------|
                                                100: SP-> |      v3       | -6
          |---------------|                               |---------------|
102: SP-> |      v3       | -6                  102:      |      v2       | -4
          |---------------|                               |---------------|
104:      |      v2       | -4                  104:      |      v1       | -2
          |---------------|                               |---------------|
106:      |      v1       | -2                  106: FP-> |    古いFP     |  0
          |---------------|                               |---------------|
108: FP-> |    古いFP     |  0                  108:      |リターン・アドレス| +2
          |---------------|                               |---------------|
110:      |リターン・アドレス| +2               110:      |               | +4
          |---------------|                               |---   p1   ---|
112:      |      p1       | +4                  112:      |               | +6
          |---------------|                               |---------------|
114:      |      p2       | +6                  114:      |      p2       | +8
          |---------------|                               |---------------|
116:      |      p3       | +8                  116:      |      p3       | +10
          |===============|                               |===============|
118:      |    古いFP     |                     118:      |    古いFP     |
          |---------------|                               |---------------|
120:      |リターン・アドレス|                  120:      |リターン・アドレス|
          |---------------|                               |---------------|
122:      |     argv      |                     122:      |     argv      |
          |---------------|                               |---------------|
124:      |     argc      |                     124:      |     argc      |
```

図 4・10 型の不整合によってできたスタック

てのオフセットは正しくない．そして，その間違ったものに続く引数は正しくアクセスできなくなる．この状態が図 4・10 に示されている．ここでは，フレーム・ポインタからの負のオフセットは正しい（期待どおりに，ローカル変数 v1，v2，v3 がさし示される）．しかし，正のオフセットは正しくない．パラメタ p2 をアクセスするのに使用されるオフセット +6 では，パラメータ p1 の上位ワードをさし示している．p3 に対するオフセット（+8）で p2 をアクセスする．p3 はアクセスされない．foo( ) は前のフレーム・ポインタの値を +10 とみなしている．

スタック上に十分な数の引数がプッシュされなければ，もっと重大な問題が起こる．この状況を図 4・11 に示す．再び，foo( ) が宣言されたとき p1，p2，p3 に対するオフセットが計算される．そして，foo( ) はそれらの変数が正しいオフセットにあると予定している．もし foo( ) が引数を持たずに呼びだされたら，そのとき p1，p2，p3 はスタックにプッシュされていない．しかし foo( ) は引数がそこにないことを知らない．これらの変数が使用されないならば，その

foo( ) が期待するスタック：

```
                      フレーム・ポインタ
アドレス：            からのオフセット：

               |---------------|
102:  SP->     |      v3       |   -6
               |---------------|
104:           |      v2       |   -4
               |---------------|
106:           |      v1       |   -2
               |---------------|
108:  FP->     |    古い FP    |    0
               |---------------|
110:           |リターン・アドレス|   +2
               |---------------|
112:           |      p1       |   +4
               |---------------|
114:           |      p2       |   +6
               |---------------|
116:           |      p3       |   +8
               |===============|
118:           |    古い FP    |
               |---------------|
120:           |リターン・アドレス|
               |---------------|
122:           |     argv      |
               |---------------|
124:           |     argc      |
```

foo( ) が引数を持たないで誤って呼ばれたとき，つくられたスタック：

```
                      フレーム・ポインタ
アドレス：            からのオフセット：

               |---------------|
102:           |               |
               |---------------|
104:           |               |
               |---------------|
106:           |               |
               |---------------|
108:  SP->     |      v3       |   -6
               |---------------|
110:           |      v2       |   -4
               |---------------|
112:           |      v1       |   -2
               |---------------|
114:  FP->     |    古い FP    |    0
               |---------------|
116:           |リターン・アドレス|   +2
               |===============|
118:           |    古い FP    |   +4
               |---------------|
120:           |リターン・アドレス|   +6
               |---------------|
122:           |     argv      |   +8
               |---------------|
124:           |     argc      |
```

図 4・11 十分に引数が入れられなかったスタック

変数がなくても問題ではない．しかし，もし，パラメータp2がfoo( )で更新されるとしたら，コンパイラはフレーム・ポインタから+6のオフセット（p2が本来あるべきところ）にあるメモリを更新する命令コードを生成する．正しい方（左）のスタックをみれば，正しい位置にあるp2をみることができる．間違った方（右）のスタックをみると，フレーム・ポインタから+6のオフセットには呼出しをするサブルーチンに必要なリターン・アドレスを保持している．これが変更されると，foo( )が正常に動作しreturnを行った（呼びだしたほうのサブルーチンも正しく動作した）とし，呼びだしたほうのサブルーチンがreturnしようとするとき，リターン・アドレスはfoo( )によって上書きされていて，呼びだしたサブルーチンはメモリの他の場所にあるように指示されている．これは，バグが実行されてからでないとわからないから，発見することがかなり難しいバグである．異常の場合は，スタックにあるものが何でもでたらめに変更されてしまう．モラルとしてサブルーチンに渡す引数の数をかぞえ，型を確かめるべきである．または，lint（読者よりは間違いを起こさない）を使用する必要がある．

最近のCコンパイラの多くは，ファンクション・プロトタイピングというサブルーチンに渡す引数の型と数をチェックする機構をサポートしている．ファンクション・プロトタイピングは，型の並びを含んだextern文で起動される．例えば

```
extern double   foo( int, char*, long );
```

はfoo( )がdoubleをかえすサブルーチンであることをコンパイラに知らせる．これは，int，charへのポインタ，longの型の3つの引数を持つ．コンパイラは，指示されたパラメータ以外のものでfoo( )を呼びだしているのをみつけると警告メッセージを生成する．不定数の引数を持った関数（printf( )など）は，型の並びの後に省略符号をつけて規定する．

```
extern void     printf( char*, ... );
```

ここで，printf( )は最初の引数としてcharへのポインタをおく．その後にはさまざまな型の引数が続く．ファンクション・プロトタイピングは，デバッグの時間を減少させる．もし，読者のコンパイラがサポートしているのなら，それを使うことをおすすめする．extern文は，たとえ同じファイル内にサブルーチンの宣言があったとしても，ファンクション・プロトタイピングを動作させるのに必要である．

最後のスタックに関連した問題は，スタック・オーバーフローである．いままでみてきたように，すべてのauto変数はスタック上に維持される．ローカルな配列用のメモリもはっきりとstaticを宣言していなければ，スタックから割り当てられる（すべてのstatic変数はスタック上ではなく，メモリに固定して割り当てられる）．10 Kバイトのautoの配列が，サブルーチンに宣言されていると，そのサブルーチンのスタック・フレームは，スタックを10 Kバイトより少し越えて使用するだろう．一般には，スタックに10 Kバイトも使えない．まずいことにコンパイラは通常このことを知らない．結果として，スタックはよくプログラムのデータ領域，そしてひょっとすると命令コードの領域にも拡張してしまう．その配列の内容を変更すると，十中八九他のルーチンのデータや，または悪くすると命令コードを変更することになるだろう．繰りかえしいうけれども，このバグは，悪さをしたサブルーチンからリターンした後長いこと現れない，それで，やはり発見することが難しい．原則は，大きな配列は全部staticにするか，malloc( )とfree( )を使って専用のメモリを得ることである．

読者はいまごろ，スタック・フレームは危険なのに，どうして使うのかと自問しているところかも知れない．使用するのにはいくつか理由がある．第1に，前にいったように，スタック・フレームを使うと異なる変数に同じメモリを再利用できるので，効率的なRAMの使用ができる．必要とする最大のスタックの大きさは，入れ子になるサブルーチンのもっとも上のレベルに対応する．第2に，スタック・フレームなしでは，再帰呼出しのサブルーチンを書くことはできない．8章で，再帰呼出しのルーチンがどのようにスタックを使うかより詳しくみるだろう．最後に，多くの引数がスタックにどのようにおかれているか，そしてその引数の大きさをサブルーチンに知らせることができれば，不定数の引数を持つサブルーチンを書くことができる．9章では，不定数の引数を持つprintf( )が，どのようにこの作業を行うか調べる（引数の数を決定するために，フォーマット中の%記号の数をかぞえ，型を決定するための変換文字を調べる）．

スタック使用の別の結果を図4·9に示す．コード生成に関しては，main( )も他と同様にひとつのサブルーチンである．main( )に特別なものは何もない．main( )は，foo( )のようにスタック・フレームを設定し，そしてもとにもどす同じコードを生成する．もっとも厳密にいえば，main( )はスタック・フレームを必要としない（実際，main( )についてひとつだけ特別なことは，リンカが

main( ) の存在を要求することである).

main( ) のスタック・フレームの一部は引数 agrv と agrc である．これらは，root モジュールから渡される．main( ) が，argv と argc を使用しないつもりならば，それらは main( ) のパラメータ・リストに宣言する必要はない．引数のプッシュとポップは，呼出しをするルーチンの仕事なので，外部の引数の存在は全く問題にはならない．いいかえれば，サブルーチンが呼びだされる（そして main( ) がそのサブルーチンである）とき，その引数はスタックにプッシュされている．コンパイラは，ただそのサブルーチン呼出し文の一部である引数の並びをみて，必要なプッシュ命令を生成する．引数はスタック上にあり，呼びだされたルーチンに使用されるかされないかには関係ない．しかし，呼びだされたルーチンは，そのサブルーチン宣言の一部である引数並びに，それらの引数がなければ，引数を得ることはできない．コンパイラが，サブルーチン宣言の正式な引数並びを処理するときに，フレーム・ポインタからのオフセットを計算する．呼出しのときではない．サブルーチンの呼出しと，サブルーチン宣言の間にはコンパイル時のつながりはない．つながりは実行時までできない．

スタック・フレームに関する問題とまとめてみると

（1）　スタック上にプッシュされた引数が，正しい型であるのか呼びだされたサブルーチンは知る方法がない．

（2）　呼びだされたサブルーチンは，呼出しをしたルーチンが正しい数の引数をスタックにプッシュしたのかわからない．ただ，引数がちゃんとしたところにあると仮定している．

（3）　ルーチンは，引数がスタックにいくつあるか，その型は何か，知ることができれば，不定数の引数を渡されることができる．

4.3　サブルーチン呼出しにキャスト（cast）を使う

変数を特定の型にしておくことは，その変数をサブルーチンに渡すときには必ずしも都合がよくない．それで，C はキャストという有効な演算子を与えている．キャストは，変数や定数をサブルーチンへ受け渡すようなときのために，その変数や定数を他の型に変換するようコンパイラに指示する．キャストは，かっこで囲まれた型の定義のようなものである．キャストはどのように行うか示すと，まず希望の型の変数宣言（後にセミコロンはつけない）を書き，その定義をかっこ

で囲む．そうして変数名を消し，キャストで変換したい変数の名前を後に書く．

例をみてみよう．

```
foo( longvar )
long    longvar;
{
        /* ... */
}

main( )
{
        int     intvar;

        /* ... */

        foo( intvar );  <------ 誤り
}
```

ここで，main( ) は foo( ) を呼びだして intvar を渡したい．intvar の方は int である．しかし，foo( ) は型 long を期待している．示したように foo( ) への呼出しを実行すれば，さっき述べたスタックにアライメントの間違いの問題をつくる．この問題は，intvar を long にキャストすることで回避できる．キャストを書くには，まず long の通常の定義を書く．

```
long    x;
```

次に，セミコロンを消してその定義をかっこで囲む．

```
(long x)
```

最後に，変数名を消す．

```
(long)
```

それで，たった今つくったキャストを，invar の前におくことによって，（スタックにプッシュする前に）invar を long に変換するようコンパイラに知らせることができる．

```
foo(  (long) intvar );
```

コンパイラによっては，int と long は同じサイズのものもある（例としては，VAX と 68000 のコンパイラの多くは int も long も 32 ビットである）．さきのプログラムは，これらの計算機ではキャストなしでも正しく動作する．しかし，他の計算機に（同一計算機の他のコンパイラに）移動しようとしても，動作しないだろう．読者のプログラムを互換性のあるものにするつもりならば，たとえいまは不必要なものに思えてもいつもキャストを使うべきである．6 章で，ポインタについて述べるときに再びキャストについて検討する．

4.4 返り値 (return value)

スタック・フレームに関連した問題のひとつに，サブルーチンの返り値がある．返り値は，通常レジスタで呼出しをしたルーチンにもどされる．呼出しをしたルーチンは，このレジスタには有効な値が入っていると想定している．すべてのサブルーチンは値をかえす，しかし，その値は，明示されたreturn文で正しくレジスタに入れられていなければ，ごみがかえされる（呼出しをしたサブルーチンがこの値を使わなければ，この値がごみであっても問題はない）．こういう理由で，複数のreturn文を使うことはよい考えではない．サブルーチンは値をかえすならば，明示したreturn文を経由しないそのサブルーチンからの他の通路がないか確かめるべきである．

4.5 シンボル・テーブル (symbol table)

コンパイラは，メモリ内の変数の位置を記録するために，シンボル・テーブルを使用する．静的，またはグローバル変数の場合は，絶対アドレスのメモリにある．ローカル変数の場合，それらはフレーム・ポインタからの，あるオフセットの位置にある．シンボル・テーブルはコンパイル時に使用される．シンボル・テーブルの要素は，コンパイラが宣言にあったときに登録され，変数がもはや必要なくなったときに削除される．例えば，ローカル変数が宣言されたときにテーブルに入れられ，コンパイラがサブルーチンを処理し終わったら削除される．図4·12にシンボル・テーブルを示す．これはコンパイラが図4·1のプログラムを処理しているときの様子である．

	変数名	クラス	型	値
1:	p1	固定	int	500
2:	p2	固定	int	502
3:	p3	固定	int	504

図 4·12 図4·1の行Bを処理中のシンボル・テーブル

コンパイラは，3つの宣言しかみなかったので，3つの変数だけが示されている．クラスは変数の記憶クラスである．変数が固定したアドレスにおかれていれ

ば固定，スタック上におかれていれば自動，外部（異なるファイルで宣言されている）にあればexternと書かれている．型は変数の型（char，int，long 等）．サブルーチンの方は，サブルーチンの返り値が適用される．値は，メモリの絶対アドレスか，フレーム・ポインタからのオフセットである．これは，記憶クラスによって異なる．自動変数はオフセットを使い，固定した変数は絶対アドレスを使う．サブルーチンの値は，サブルーチンの最初の命令のアドレスである．現実のコンピュータのシンボル・テーブルは，ここに示すよりもっと複雑である．しかし，基本的なものはすべてこの例の中にある．

コンパイラが行Eを読むまでに，いくつか変数がテーブルに加えられている．新しい状態を図4·13に示す．

```
           変数名           クラス     型        値
     -------------------------------------------------
     |                    |          |        |          |
 1:  |     p1             |   固定   | int    |   500    |
 2:  |     p2             |   固定   | int    |   502    |
 3:  |     p3             |   固定   | int    |   504    |
 4:  |     foo            |   固定   | int    |   506    |
 5:  |     p1             |   自動   | int    |   +4     |
 6:  |     p2             |   自動   | int    |   +6     |
 7:  |     p3             |   自動   | int    |   +8     |
     +--------------------+----------+--------+----------|
```

図 4·13 図4·1の行Eを処理中のシンボル・テーブル

サブルーチンであるfooは，固定したメモリアドレスにある．その型は，返り値の型である．いま，変数p1，p2，p3に対して3セット登録されている．その2番目の登録（テーブルの5～7行目）は，foo(　)の引数並び（行CとD）にある3つの変数である．コンパイラが変数を使用するときには，テーブルを下から上に探し，該当する名前を持った最初にみつけた登録要素を使用する．foo(　)の中でパラメータp2が使用されていれば，コンパイラはテーブルをさかのぼってp2を探し，テーブルの6行目でp2を発見する．そのときローカル変数p2は，同じ名前のグローバル変数より優先される（なぜなら，コンパイラは該当する名前を持った変数をみつけたら探すのをやめるから）．C言語では，すべてのローカル変数は，同じ名前のグローバル変数より優先する．

さて，もう少しプログラムを処理する．図4·14は，foo(　)のすべてのローカル変数を処理した後のシンボルを示す．

	変数名	クラス	型	値
1:	p1	固定	int	500
2:	p2	固定	int	502
3:	p3	固定	int	504
4:	foo	固定	int	506
5:	p1	自動	int	+4
6:	p2	自動	int	+6
7:	p3	自動	int	+8
8:	v1	自動	int	-2
9:	v3	自動	int	-4
0:	v3	自動	int	-6

図 4・14　図 4·1 の行 G を処理中のシンボル・テーブル

ここで，本当のローカル変数（引数と比較して）は，フレーム・ポインタからのオフセットが正数ではなく負数になっている．コンパイラが，foo( ) の処理を終えたとき（つまり，行 I の '}' をみつけたとき）テーブルから foo( ) のすべてのローカル変数の参照が削除される．それを図 4·15 に示す．

	変数名	クラス	型	値
1:	p1	固定	int	500
2:	p2	固定	int	502
3:	p3	固定	int	504
4:	foo	固定	int	506

図 4・15　図 4·1 の行 J を処理中のシンボル・テーブル

foo( ) 自体（つまりサブルーチン名）がグローバル変数として扱われる（グローバル変数と同じレベルで宣言されているから．すなわち，どのブロックにも含まれていないから）ことに注意しなさい．main( ) が処理されるとき，テーブルは図 4·16 のようにみえる．main( ) はローカル変数を持たないので，テーブルはこれ以上大きくならない．

ここで注意するべき重要なことがいくつかある．第 1 に，変数の参照はシンボル・テーブルを通じて行われる．サブルーチンは，呼出しをしたルーチンが，実際にすべてのパラメータをスタックにプッシュしたかどうか知る方法はない．正規の引数並びに，いくつかのパラメータが宣言されていることがわかるだけである．

	変数名	クラス	型	値
1:	p1	固定	int	500
2:	p2	固定	int	502
3:	p3	固定	int	504
4:	foo	固定	int	506
5:	main	固定	int	600

図 4・16 図4・1の行Lを処理中のシンボル・テーブル

第2に，変数についてのすべての情報はシンボル・テーブルを逆に探して決定される．たとえ他にもテーブル内に同じ名前の変数があったとしても，最初にみつかった変数が使用される．

最後に，サブルーチン名は，その宣言に出会うまではシンボルテーブルに登録されない．コンパイラが，サブルーチンの返り値を知る唯一の方法は，その宣言を処理することである．したがって，サブルーチンを宣言するまえに使うと，コンパイラはシンボル・テーブルにそのサブルーチン名をまだ登録していない．サブルーチンを使う前に，必ずサブルーチンの宣言をするようにプログラムを書くことはいつも都合がよいわけではないから，コンパイラは少し仮定をする．int をかえす extern サブルーチンであるとして，シンボル・テーブルに宣言されていないサブルーチン名を入れる．そのサブルーチンが，同じファイル内に後で宣言されていてもたいしたことではない．リンカがそれを発見するだろう．

4.6 制御の流れ：if/else, while 等

図4・17に，if/else をアセンブリ言語に変換したものを示す．x と y を static にしたのでプログラムは簡単である．条件（x>y）は，3つの命令を必要とする．x の値を A レジスタにおき，y の値をひく（A ＝ x － y）．A レジスタの前の内容は破壊される．x が y より大きければ，結果は正で P フラグはひき算の結果として1にセットされる．x が y に等しければ，結果は0でZ フラグが1にセットされる．x が y より小さければ，結果は負でN フラグが1にセットされる．JLE 命令は，N か Z フラグが1にセットされているときだけジャンプする．if 文の条件部分に規定されたのと反対の条件で試して，その反対の条件が真だったら else 節の方にジャンプする．つまり，x>y で調べて，x＜＝y のときだけ

```
static int      x;        |   x:      DW      1
static int      y;        |   y:      DW      1
                          |
if( x > y )               |   if:     MOV.W   x,A
{                         |           SUB.W   y,A
                          |           JLE     else
                          |
      <処理>              |             <処理>
                          |
}                         |           JMP     next
else                      |   else:
{                         |
      <処理>              |             <処理>
}                         |
                          |   next:
```

図 4・17　if/else 文

分岐する（JLE 命令を使用する）．この入り組んだプログラムの仕方によって，3 つ目の JMP 命令を使わないですむ．if 節は，else 節の前の無条件ジャンプで終わる．

図 4･18 は，while ループを示す．前の例で述べたように，ひき算をしてから反対の条件でループの外にジャンプさせて判定する．ひとつ違うことは，条件に合わないとき eles 節にジャンプする代わりに，ループの外にジャンプすることである．while ループの continue 文は，JMP while 命令に変換される．while ループの break は，JMP next に変換されている．

図 4・18　while ループ

for 文は，図 4･19 に示されている．for のループ部分を処理するために生成された命令コードは，while のものと同じであることに気づくだろう．初期化と増加させる命令コードが for の場合には加えられている．しかし，さきの例の while

```
static int      x;
static int      y;


for( x = 0; x > y; x++ )
{

       <処理>



}
```

```
x:        DW        1
y:        DW        1

          MOV.W     #0,x
for:      MOV.W     x,A
          SUB.W     y,A
          JLE       next

             <処理>
endfor:
          ADD.W     x,#1
          JMP       for

next:
```

図 4・19 for ループ

文の前にx ＝ 0をおき，ループの終わりにx＋＋をおけば同じ命令コードが生成されただろう．通常は，ソース・プログラムを読みやすくする制御文を使う．目的コードは，多分同じになるだろう．

for ループの continue 文は，JMP endfor 命令を生成する．一方 for ループの break は，JMP next に変換される．

文 for(；；) と while(1) は "do forever" としてよく使用される．多くのコンピュータでは，図 4・20 に示すように，for ループに対してもっと効率のよい命令コードを生成する．

```
while( 1 )                  while:   MOV.W    #1,A
{                                    OR.W     A,A
                                     JZ       next

       <処理>                            <処理>

                                     JMP      while
}                           next:



for( ;; )                   for:
{
       <処理>                            <処理>

}                                    JMP      for
```

図 4・20 無限ループ

while(i) は，while(i ！＝ 0) と比べて同じだと考えられる．！＝と明示すると不必要な命令を生成させることになる．

4.7 switch

switch は if/eles よりは複雑である．switch は，ソースコードをみても，コンパイラが何をしようとしているのかよくわからない，C 言語の 2，3 ある要素のひとつである．さらに，コンパイラは，異なる状況では違ったことをしようとする．例として Lattice C コンパイラのバージョン 2.14 用のマニュアルを引用する（pp. 4-26 f.；かっこ内は著者のコメントである）．

> コード生成は，switch 文には効率的な命令コードを生成しようと特に努力する．case の値の数と範囲によって，3 つの命令コード列のうちひとつが生成される．
>
> （1） case の数が，3 かもっと少なければ，制御は条件と分岐命令で case の要素へとぶ（つまり，switch は if/else の繰りかえしのように扱われる）．
>
> （2） case の値がすべて正で，最大の case と最小の case の差が case の数の 2 倍より少なければ，コンパイラは，switch の値で直接指示される分岐表を生成する．値は，必要ならば，最小の case の値に調節されて，指標づけするまえに表の大きさと比較される．この構造は，最小の実行時間と，3 番に記述されるタイプのものに要求されるのより小さい表を必要とする．
>
> （3） 上記以外であれば，コンパイラは [case 値と分岐アドレス] が対になった表を生成する．これは，switch の値を順に探す（この処理は，よく使用される case が表の終わりにあると，特に，非効率である）．

コンパイラの中には，すべての switch を最適化しないで，（3）の場合として扱うものもある．これは switch をひどく非効率的にしてしまう．他のコンパイラには，バイナリ・サーチができるように case 値の表を再構成して，（3）の場合をもっと効率よくしているものもある．このように，switch の最初の case 文の位置は，case が最初にみつかったところであるとは限らない．しかし，if/else の if 文は，いつも最初に評価される（case が評価される順序と，計算機が該当する case に制御を移した後の命令コードが実行される順序とを混同すべきではない）．

これらすべてが意味することは，switch はとても非効率的になりうるということである．ひとつか 2 つ else 節があるくらいで，if/else の代わりに switch を使用するべきではない．その一方で，switch は長々と続く if/else よりもしばし

ば読みやすいプログラムができる．

4.8 再配置可能プログラム（relocatable code）

明確にするために，いままで述べてきたプログラムは，すべて再配置可能ではなかった．再配置可能プログラムは2つの属性を持っている．メモリ中のどこでも走らせられなければいけない．そして，他のモジュールとリンク可能でなければならない．

プログラムを再配置可能にするには2つの方法がある．8085のような計算機では，絶対アドレスのジャンプしかサポートしていない（つまり，JMP 100とは書けてもJMP 124(PC）とは書けない）から，仮アドレスの仕組みを使う．コンパイラは，コンパイル時にジャンプに目的のアドレスを与えることができない（なぜなら，コンパイル時に，処理中のモジュールがメモリのどこにおかれるかわからないから）．したがって，コンパイラがアセンブリ命令をつくるときに，通常は目的のアドレスを入れる場所に，アドレスそのものではなく，目的のアドレスへのオフセットを入れる．このすべてのオフセットがプログラムに入れられた位置を示す表を保持している．外部オブジェクト（このモジュールに宣言されていないサブルーチンやデータ）と，外部オブジェクトがプログラム中で使われている位置を示す別の表がある．その位置は，通常外部オブジェクトをアクセスする各命令へのそのモジュールの先頭からのオフセットである．3番目の表は，外部からアクセスすることができる，現在処理中のモジュール内にあるすべてのオブジェクトを記載している．予約語staticは，グローバル変数に使うと，その変数を3番目の表に含ませない．したがって，静的グローバル・オブジェクトは，リンカがアクセスすることはできないので，そのオブジェクトが宣言されているファイルの外からアクセスすることはできない．リンカは，最終的なプログラムにまとめるときに，もとのプログラムの不完全な部分を解決するためにこれらの表を使用する．リンカは，外部オブジェクトへの参照を，その実際のアドレスと置き換える．そして，そのときはモジュールのベース・アドレスはわかっているから，ジャンプ命令のオフセットも同様に絶対アドレスに置き換える．リンカへの入力となる中間モジュールは，再配置可能である．しかし，最終的な（リンクされた）プログラムは，再配置可能ではない．メモリ中の特別な領域でだけ走ることができる．

真の再配置可能なプログラムは，相対アドレスをサポートしている計算機(68000やPDP-11のような）上でだけ行われることができる．この場合，絶対アドレスへのジャンプや絶対アドレスのメモリ・アクセスなどはつくらない．むしろ，すべてのジャンプは現在のプログラム・カウンタと相対的に行われる［JMP x(PC)］．外部サブルーチンへは通常，2回のジャンプでアクセスされる．命令コードは，モジュール内の既知の位置にあるジャンプ・テーブル（ジャンプ命令の長い表）へのジャンプが生成され，それから，リンカがプログラムをリンクするとき適切な相対アドレスを含むジャンプ・テーブルに修正する．外部データは，データ領域（リンカによって供給される）のベース・アドレスに相対的にアクセスされる．このように，2番目の形態のプログラムはメモリ中のどこにでも配置することができる．

4.9 練　　習

4-1　前章で述べたアセンブリ言語を使用して，次のソース・プログラムをみてコンパイラが生成するようなアセンブリ言語プログラムを書きなさい．intは2バイト占めるとする．呼びだされた関数の返り値はレジスタDで呼出しをした関数にかえされる．

```
test( var1, var2 )
int var1, var2 ;
{
        int     x, y;

        x = var1;
        y = var2;

        while( x < y )
                x = x + y;

        if( x >= 3 )
                return( x + y - 1 );
        else
                return 0 ;
}

main( )
{
        static int a, b;

        a = 2;
        b = 3;

        test( a, b );
}
```

4-2　練習4-1にあるサブルーチンtest( )の，スタック・フレームの図をかきなさい．スタック・ポインタとフレーム・ポインタ，そしてフレーム・ポインタからの関連するオフセット全部を示しなさい（図4·9のように）.

4-3　上記の図を使用して，次の処理の後シンボル・テーブルがどのようにみえるか示しなさい．

(a)　test( )にあるint x，y；が処理された．

(b)　main( )モジュール中のtest( )への呼出しが処理された．

4-4　読者のコンパイラに使用されているスタック・フレームがどのように構成されているか記述しなさい．auto変数と，サブルーチンの引数がどのようにアクセスされるか説明しなさい．読者のコンパイラが，サブルーチンのアセンブリ言語のダンプをつくるのならば，それを含めること．

5 構造化プログラミングとステップワイズ・リファインメント

構造化プログラミングとは，組織だった維持しやすいプログラムを書くためのテクニックに対してつけられた一般的な名称である．本章では，構造的な方法でのプログラムの書き方をみる．さらに，書式やプログラムの編成の仕方，そしてコメントづけのような周辺の問題も検討する．本章は，構造化プログラミングの話題には少しだけ触れて，多くは構造化プログラミングのテクニックが，小さなプログラム（2 000 行以下）を開発するための使われ方について述べる．この話題は読者が思うよりも役にたつだろう．

構造化プログラミングのテクニックは，ときどき，"goto 文のないプログラムを書くこと" といわれている．しかし，goto 文を使うか使わないかは，よく構造化されたプログラムを書くのにはたいした問題ではない．goto 文があってもよいプログラムを書くことができるだけでなく，goto 文がないプログラムより読みやすかったりもする．構造化プログラミングは，思考過程を整理してプログラムを開発するひとつの方法である．そのために，プログラムは開発時間が短くてバグが少ない．

残念なことに，多くの学校では，この構造化プログラミングの授業を，情報処理教育課程以外の学部生徒に使っている．結果として，構造化プログラミングという言葉は，サークルによっては悪い評判を得てきた．多くの人が，実用的でない机上の練習のような構造化プログラミングを知る．その人達は，構造化プログラミングをただのたいへんな作業の形のかわったものとしてとらえ，学部生徒の生活をより忙しくすることを意味している．真理にまさるものはない．プログラムが構造的な方法で開発されるとき，多くのプログラムは，より少ない時間で書

かれ，もっと信頼性を高くすることができる．

構造化プログラミングのテクニックは，C言語でプログラムするときには欠くことができない．言語の中には，PascalやModula-2のように，言語の構文としてプログラム構造を組み込んでいるものもある．構造化されていないPascalプログラムを書くことは（書こうと思えばできないことはないけれども）難しい．一方で，C言語は，言語自体に組み込まれた厳格な構造はない．C言語は，プログラム編成と開発方法に大きな自由度がある．したがって，C言語に加えて構造化プログラミングも学ばなければ，自分をプログラムの穴に落として，動かず，直すこともできない40 000行のプログラムの中で自分を探すことになるだろう．

5.1 プログラムを書くことは作文である

プログラミングと数学が，どこか関係があるというのは一般的な誤解である．ここでは，情報処理科学（computer science）とプログラミングとを，すなわちプログラムの研究とプログラムを書くこととを区別している．情報処理科学の多くは，数学的な解析をしなければならない．一方，数学者でなくとも，よいプログラマになることは可能である．数学でないならば，何がプログラミングの基礎を与えるのだろうか？　プログラムを書くテクニックは，随筆を書くテクニックとほとんど同じである．英作文の授業で学んだ技術が，プログラミングに応用されるとき，とても価値がある（多くの学校で，情報処理科学の単位として作文を（数学と同様に）必要とすれば，プログラムがもっとよく書けるばかりでなく，読みやすいドキュメンテーションも書けるだろう）．本章では，構造化プログラミングについて話すための手段として，随筆を書くメカニズムを使うつもりである．話がすすむにつれて，その2つの過程を比較する．

随筆を書く最初の，もっともたいへんな段階は題目を決めることである．広い分野を，有効なスペース内で，十分に議論することのできる主題に狭める．いったん題目を選んだら，それについてよく調べ，他の人のいったことを学び，そして他の人の仕事と，もとの起源を調べて自分自身の結論を書く．要約すると，随筆を書く最初の段階は考えることである．

不幸にも，考えることはプログラム開発ではたいてい忘れられている段階である．問題の解決方法に論点を絞るまえに，随筆の題目を書くことができる範囲に狭めなければならなかったように，その問題を正確に定義しなければならない．

プログラムを書きながら，問題を明確にしようとするのは誤りである．どの未解決の問題も事前に考えられていなければいけないし，不慮の出来事も考慮されていなければならない．もちろん，この段階ですべてを考えることはできない．しかし，試みることで膨大な作業から自分自身を救うことができる．そうしなければ，プログラムを書くことに時間を費やすことになるだろう．その場合，仕事は進まないし，修正することも困難になるので，そうならないようにするべきである．

問題が定義されたら，その問題についてよく調べ，他のプログラマが同様の問題をどう解いたか調べる．この段階もよくとばされる．プログラマは，しばしばもうすでに解決されている問題に何時間も費やす．この時間は，以前の解決法を調べることを面倒がらなければ必要なかっただろう．

関連した問題だが，多くのプログラマは他の誰かが書いたプログラムを使用することを嫌がるということがある．そんな方法は嫌だというだけですでにできているプログラムを無視しようとしないこと．確かに，プログラムの中にはひどくまずい形式で書かれていたり，しばしば多くの人に修正され，使わない方がよかったり，草稿から始めたほうがベストであることもある．しかし，動作しているプログラムを見捨ててしまう前に，公平に，同じ程度のプログラムを自分で書くとどのくらいの時間がかかるのか，そして自分のプログラムのほうがそれより本当によくなるのか算定しなければならない．

随筆の中で述べることを決めたら，実際にそれを書かなくてはならない．同様に，プログラムの課題を決めたら，そのプログラムを書かなければならない．たとえすわって随筆を書き始めたとしても，思考のつながりのないページで終わったら，随筆というより，とりとめのない読みにくい雑文になってしまうだろう．同様に，ただすわってプログラムを書き始めることはできない．まずプログラムの構成を考えなければならない．随筆を書くときも，プログラムを書くときにもアウトラインを決めることが第1ステップである．実際，普通のアウトラインの形式がプログラム設計によく適している．

アウトラインを書くにはどうするのか？　整然とした形で考えを展開させる．書こうとしていることを2，3の簡潔な文で要約し，とても高いレベルで問題を定義することから始める．これらの文を1行にひとつずつ紙に書き，各文にローマ数字を割り当てる．プログラム設計に使われる過程も同じである．数行の簡潔

な文で，問題をどのように解くつもりか要約する．実際それをやってみなさい．文章にして紙に書きなさい．問題をことばに置き換えることは大変重要である．それを文章にできないのなら，C 言語で表すこともできないであろう．

短いけれども具体的な例をみてみよう．次のサブルーチン（8 章にある）が与えられたとする．

```
int     parse( expr )
char    expr[ ];
```

このサブルーチンは，算術式を含む文字列を入力とし，その式を評価し整数の結果を出力する．このプログラムぐらいの，小さな計算プログラムをつくりたい．

プログラムを書くまえに，そのプログラムがするべきことを定義しなければならない．この定義は，随筆の題目を段落に分けることに等しい．記述は簡潔だが可能なかぎり完全であるべきである．次の各段落は記述として好ましい．

> 8 章にあるサブルーチン parse( ) を動作させるプログラム expr を書きなさい．そのプログラムは，算術式を解析して結果を標準出力にプリントする．式は 10 進数で，かっこと加算，減算，乗算，除算に対する +，−，*，/ からできている．式中に空白とタブは認められない．インフィックス記述[訳注 2]（infix notation）が使用される．
>
> プログラムの呼出しは 3 つの形式がある．
>
> `expr`
>
> は標準入力から 1 行に式 1 づずつ連続して得て，それらの式を評価し，結果を標準出力にプリントする．すべての式は改行で終わり，それは式の一部とはみなされない．
>
> `expr <exp1> ... <expN>`
>
> はコマンドから式（expN）を抜きだして，それらを評価し，式とその結果を標準出力にプリントする．各引数[訳注 3]には，式がひとつだけあり，引数の式は完全なものでなければならない．
>
> `expr -f <ifile>`
>
> は，入力がコマンドからでなく，指定されたファイルからである以外は，前の形式と同じように動作する．そのファイルは 1 行につきひとつの式があ

訳注 2：通常の計算式の書き方である．（例）10+2=12
訳注 3：式（expN）は expr を呼びだすときの引数

る（最初の例と同様）.

プログラムは入力した式を，計算した結果といっしょに，標準出力にプリントするべきである．式が評価できないときにはエラーメッセージをだす．繰りかえしモードで式が演算子で始まっているときは，前の評価の結果を現在の式の最左の項として使う（式の最初の項として，負数を使う必要があるときは（−1）を用いる）.

次の段階は，データがどのように表されるかを決めることである．データの型の選択が，しばしばプログラム全体の構成に影響する．それで，この決定をはやくすることがもっともよい．この例は，2つのタイプのデータを必要とする．式を保持するためのバッファと，解析結果を保持するための変数である．2つの #typedef と #define でデータを定義する．これは，expr.h というファイルに入れられる（図5·1参照）.

```
#define MAXBUF   128

typedef char     BUFFER[ MAXBUF ];
typedef int      ANSWER;
```

図 5·1　Expr.h

このファイルは，プログラム中のすべてのファイルに含まれている（プログラムが複数ファイル構成になった場合）．このプログラムで使用されているすべての定数は，マクロにするべきである（図5·1のMAXBUFのように）．このようにすると，入力バッファの大きさをかえるときには，MAXBUF マクロを訂正し，数字で書いた定数を探すために，すべてのソースファイルを調べなくても再コンパイルすることができる．

第3の段階は，記述したものを単純な動作に書き直し，これをアウトラインの形式にすることである．プログラムのアウトラインは図5·2に示されている．

```
I.  もしコマンド行に引数がないならば
        A.  式を会話的に処理する

II. その他（コマンド行に引数がある）
        A.  もし最初の引数が −f であるならば
            1.  ファイルからの引数を処理する
        B.  その他
            1.  コマンド行からの引数を処理する
```

図 5·2　簡単なプログラムのアウトライン

もちろん，このアウトラインではとても短い随筆と，とても短いプログラムになる．しかし，アウトラインにはまだプログラムを書くには十分な情報がない．図5･3のように，アウトラインを少し詳しくする．

Ⅰ．もしコマンド行に引数がないならば
　A．式を会話的に処理する
　　1．開始のメッセージをプリントする
　　2．入力がある間
　　　a．プロンプトをプリントする
　　　b．入力行を得る
　　　c．式を評価する
　　　　i．もし式が演算子で始まっているならば
　　　　　(a) 前の演算結果をその式の最左の項として使用するために式の文字列を修正する
　　　　　(b) 式を解析する
　　　　　(c) もしエラーがあったら
　　　　　　(i) エラー・メッセージをプリントする
　　　　　(d) その他
　　　　　　(i) 結果をプリントする

Ⅱ．その他（コマンド行に引数がある）
　A．もし最初の引数が －f であるならば
　　1．ファイルからの引数を処理する
　　　a．ファイルをオープンする．もしオープンできなかったらエラー・メッセージをプリントする
　　　b．入力行がある間
　　　　i．式解析ができるように入力行を修正する
　　　　ii．式を評価する

　B．その他
　　1．コマンド行からの引数を処理する
　　　a．引数がある間
　　　　i．式を評価する
　　　　ii．次の引数を得る

図 5･3　アウトラインを詳細にする

式の評価は，3つの場所（I. A. 3. c.，I. A. 1. b. ii，II. B. 1. a. i）で行われることに気づくだろう．今，同じことをしているのに気づいて，ひとつのサブルーチンが，実際全部の式の評価をするために使用されるプログラムを書くことができる．前もってこのような決定をすることは，できあがったプログラムを修正するより簡単である．

随筆に話しをもどす．随筆全体を要約する段落から始めなければならない．その要旨説明(訳注4)の段落は，アウトラインの主な項目をまとめることでできるだろう．Cプログラムでは，要旨説明の段落に等しいものがmain( )サブルーチンである．ちょうど要旨説明の段落がつくられたように，main( )もアウトラ

訳注 4：topics paragraph

```
アウトライン:|  プログラム:
---------|-----------------------------------------------------------
         |
         |  main(argc, argv)
         |  int  argc;
         |  char *argv[ ];
         |  {
I.       |      if( argc <= 1 )
I.A      |          interactive_mode( );
II.      |      else
         |      {
II.A     |          if( argc==3  &&  argv[1][0]=='-' && argv[1][1]=='f')
II.A.1   |              file_mode( argv[2] );
II.B     |          else
II.B.1   |              command_mode(argc, argv);
         |      }
         |  }
```

図 5・4 アウトラインからプログラムへの変換

インから組み立てられる（図5･4参照．プログラム自体が理解できなくても心配することはない．ただ，プログラムの構造をみればよい）．プログラム中の入れ子の深さ(訳注5)が，正確にアウトライン中の入れ子の深さと対応している．

要旨説明の段落は，それ自体が小さな随筆として成り立つべきであり，同様にプログラムでもそうあるべきである．開発中のプログラムは，開発のおのおのの段階で十分に機能するべきである．したがって，プログラム開発の次の段階は，それを動作させることである．これを行うためには，stub と呼ばれるいくつか仮のサブルーチンを与えなければならない．stub はその存在を宣言するだけで何もしない．しかし，main( ) はサブルーチンを呼びだすから，それらのサブルーチンは main( ) をコンパイルしてテストするときに，存在しなくてはならない．図5･5に main( ) を動かすために必要な stub を示す．

stub は，ただ "I'm here" とプリントしてリターンする．簡単にできれば，引数もプリントする．このやり方で，有効な引数がサブルーチンに渡されたかどうかがわかる．あきらかに，この点では stub プログラムは多くのことはしない．それにもかかわらず，プログラムを十分にテストし動かすことができる（役立つことは何もしないかもしれないが，動作を行っている）．10 000 行のプログラムを書いて，それを一度にデバッグしようとするのは誤りである．代わりに，2，3行のプログラムを書き，それが動くようにして，さらに2，3行書いてまたそれが動くようにする．そしてそれを繰りかえすべきである．内容の変更は，一

訳注 5：プログラムのみた目のでこぼこの具合，インデント

```
interactive_mode( )
{
        printf("Doing interactive input\n");
}

file_mode( filename )
char    *filename;
{
        printf("getting input from <%s>\n", filename );
}

command_mode(argc, argv)
char    **argv;
int     argc;
{
        printf("doing command mode input\n");
}
```

図 5・5　main に使用される stub

度に2, 3個しかしなければ，プログラムが止まったとき，問題のある位置がいつでもわかる．

随筆にもどる．もうパラグラフを付加して随筆に肉付けをしなければならない．そのひとつひとつのパラグラフはアウトライン内の副主題のひとつの題材を取りあげる．さらに，main( ) を書くために使ったのと同じ手順で stub を展開させてプログラムを詳述することができる．この詳細にしたプログラムを図5·6に示す．

```
アウトライン   | プログラム：
-------------|-----------------------------------------------------------
             | #include <stdio.h>
             | #include "expr.h"
             |
             | /*----------------------------------------------------------
             |  *  isopr(c) は c が演算子 * + − / の中のひとつであったら 1 に評価する
             |  */
             |
             | #define isopr(c) ((c)=='+' || (c)=='-' || (c)=='/' || (c)=='*')
             |
             | /*--------------------------------------------------------*/
             |
I.A          | interactive_mode( )
             | {
             |      /*  interactive mode で式を処理する．プロンプトをプリントし，標準入力から式を
             |       *  得る，そして結果をプリントする．実行は EOF か空行にあったら終了する．
             |       */
             |
             |      BUFFER    buf;
             |
I.A.1        |      printf("Enter expression after ? or <CR> to exit program\n");
I.A.2        |      while( 1 )
             |      {
I.A.2.a      |           printf("? ");
I.A.2.b      |           if( gets(buf) == NULL || !*buf )
             |                break;
I.A.2.c      |           evaluate( buf );
             |      }
```

```
                | }
                |
                | /*----------------------------------------------------------*/
                |
II.A.1          | file_mode( fname )
                | char      *fname;
                | {
                |     /*  ファイルから得た式を評価する．そのファイルは読みだし専用モードでオープンし
                |      *  ていて，一度に1行ずつ処理される．行が '/n'（改行）で終わっていたら，改行
                |      *  は削除される．行の最大の長さは MAXBUF－1(132) 文字である．
                |      */
                |
                |     int       len;
                |     BUFFER    buf;
                |     FILE      *stream;
                |
II.A.1.a        |     if( (stream = fopen(fname,"r")) == NULL )
                |         fprintf(stderr, "Can't open <%s>\n", fname );
                |     else
                |     {
II.A.1.b        |         while( fgets(buf, MAXBUF, stream) != NULL )
                |         {
                |             /*  復帰コードを削除して式を評価する．
                |              *  空行は無視する．空行を評価しない．
                |              */
                |
II.A.1.b.i      |             if( (len = strlen(buf))  &&  *buf != '\n' )
                |             {
                |                 if( *(buf + --len)  == '\n' )
                |                     *(buf + len) = ' ' ;
                |
II.A.1.b.       |                 evaluate( buf );
       .ii      |             }
                |         }
                |     }
                | }
                |
                | /*----------------------------------------------------------*/
                |
II.B.1          | command_mode(argc, argv)
                | char      **argv;
                | {
                |     /*  argv から得た式を処理する．evaluate( ) が呼ばれる前に1回にひとつ argv は
                |      *  増加させられていなければならない（argc を減少させて相殺する）．
                |      */
                |
II.B.1.a        |     while( --argc >= 0 )
                |     {
II.B.1.a.i      |         evaluate( *argv );
II.B.1.a        |          argv++;
       .ii      |     }
                | }
```

図 5・6　stub を詳細にする

今は evaluate( ) という stub ひとつだけが必要である（図 5·7）．stub の evaluate( ) は，プログラムの残りが適切に動作するかどうかを示す値をもどさなければならない．

ときには新たな改良版をコンパイルし，そして動くようにする．また詳細を書く作業を続け，アウトライン全体を通して，一度にひとつのセクションを処理しながら小さなプログラムといくつかの stub を書き，できあがったプログラムを動くようにして，プログラム全体が終わるまでこの過程を繰りかえす．プログ

```
int     evaluate( expr )
{
        printf("Evaluating <%s>, returning 1\n", expr );
        return 1;
}
```

図 5・7　evaluate( ) に対する sutb

ラム開発のためのこの方法は，ステップワイズ・リファインメント（stepwise refiement）と呼ばれ，構造化プログラム開発の心臓部である．その主な利点は，プログラムがどの段階でも動作し，それでどこか悪いときにも必ず問題を発見できることである．

随筆を書くとき一度にひとつの段落を書くように，一度にひとつの stub を書くことは重要である．再コンパイルする前に，ほんの少しだけ変更することは退屈なことのように思えるかもしれない．しかし，長い目でみると時間の節約になっている（なぜなら，ぐあいが悪い時に，バグがありそうなところがいつもわかるためである．バグは変更したばかりのところにある）．

```
アウトライン：     | プログラム：
-------------------|-------------------------------------------------------
I.A.2.c            |
II.A.1.b.ii        |
II.B.1.a.1         | int     evaluate( expr )
                   | char    *expr;
                   | {
                   |      static int  def_val = 0;
                   |      int         err ;
                   |      BUFFER      copy;
                   |
                   |      /*  expr に含まれている式を評価して結果をプリントする．expr の最初の文字か演算子な
                   |       *  らば，その式の最初の項として def val を使用する　（つまり，evaluate( ) への前
                   |       *  の呼出しで生成された結果を使う）．もしエラーがあったら，結果の代わりにエラー
                   |       *  メッセージをプリントする．いずれにしても，parse( ) にかえされた値をもどす（エ
                   |       *  ラーのときは 0 である）．
                   |       */
                   |
I.A.2.c.i          |      if( isopr( *expr ) )
                   |      {
I.A.2.c.i.a        |           sprintf( copy, "%d%s", def_val, expr );
                   |           expr = copy;
                   |      }
                   |
I.A.2.c.i.b        |      def_val = parse( expr, &err );
                   |
I.A.2.c.i.c        |      if( err )
I.A.2.c.i.c.i      |           fprintf(stderr, "Error in expression: %s\n", expr);
I.A.2.c.i.d        |      else
I.A.2.c.i.d.i      |           printf("%s = %d\n", expr, def_val );
                   |
                   |      return def_val;
                   | }
```

図 5・8　evaluate( ) stub を詳細にする

ただひとつ残っているstubは，evaluate( )である．evaluate( )は図5·8に展開されている．これ以上他にはstubはない．

5.2 プログラム構成

随筆の場合と同様に，プログラムはよく構成されていると読みやすくて修正がしやすい．プログラムを構成する上でよい方法が2つある．

第1の方法は，随筆を手本にする．随筆は，2, 3行で随筆を要約する要旨説明の段落から始める．それから，順序よく話題を発展させ，関連する考えを段落にグループ分けする．段落が，考えを述べた要旨説明文で始まり，それからその考えを一連の関連した文に発展させた，この構成をうつしだしている．プログラムもこの手本に従って，サブルーチンの呼出しを連ねて，プログラムの機能を記述した上位レベルのルーチン（main( )のような）で始まる．それらのサブルーチンは，それぞれふさわしい名前をつけられている．各サブルーチンは，段落に対応し，同じようなやり方で展開する．まずそのサブルーチンから始まって，そのサブルーチンが使用するルーチンが続く．

このプログラムのやり方を手本にすると，ほとんどの作業を行う上位レベルのルーチンがファイルの先頭にあり，そして，その上位レベルのルーチンに呼びだされるルーチンがその下におかれる．サブルーチンは，下に読んでいくにつれて，より複雑になり，より下のレベルになる．モジュール内の最初のルーチンは，main( )である．

第2のやり方は，随筆全体を逆さにする．最下位レベルのサブルーチンをファイルの先頭におき，そしてその上のレベルのサブルーチンを下におく．サブルーチンは，やはり第1と同じようなやり方で開発する．しかし，開発はファイルの終わりから先頭にすすむ．このやり方の利点は，プログラム中に前方参照（まだ宣言されていないサブルーチンの使用）が少ししか要求されないことである．しかし，本来どちらのやり方が，より優れているということではない．ファイルを読みあげるか読みさげるかに関係なく，整然とサブルーチンを示すべきである．

プログラムのファイルを構成する第3のやり方は，サブルーチンをアルファベット順におくことである．前方参照の問題は，ファイルの先頭にextern文を書くことで解決される．しかし，このやり方ではプログラムはかなり読みにくい（サブルーチンと，その近くのサブルーチンとの間に密接な関係がないから）．しか

し，アルファベット順のファイルだと，特別なファイルを探すのは簡単である．

プログラムのアウトラインは，大きなプログラムを複数のファイルに分ける方法についてよい指標を与えてくれる．ローマ数字で印をつけられたサブルーチンは，それひとつが，ひとつのモジュールにおかれる．それは，アウトライン内の分類で表されたサポート・ルーチンもいっしょである．機能的に関係するサブルーチンを必要とするこのやり方は，プログラムの全体の大きさによる．しかし，この追加のモジュールもアウトラインと同様に構成される．

その他の構成上の問題は，データの位置や外部宣言などである．筆者はたいてい次の順序でモジュールを構成する．

```
#includes
extern 宣言
#defines
typedefs
グローバル変数
実行文
```

サブルーチン内に，外部変数と ♯define などを混在させないこと．サブルーチン内の定義が失われてしまうからプログラムの維持が難しくなる．

5.3　コメントのつけ方（commenting）と書式（formatting）

本章の大部分は”よい助言”の部類に入る．多くの人はコメントをつけるスタイルや書式規則について，ほとんど狂信的な情熱を持って説教する．その人達は心得違いをしている．書式の規則はプログラムを読みやすくするためのものである．そして，この目的以上のことをできるようにする．正反対の立場では，書式は一貫しているかぎりたいして重要ではないという人もいる．これは明解だが真実ではない．プログラマの書式は，いつもおそろしく大事なものになり得る．ここで述べることのいくつかは，読者は賛成できないかもしれない．しかし，本章は，この問題について読者によく理解させることができる．コメントは，すべてのプログラムにある基本的な部分である．自分自身を解説しているようなプログラムはない（よく書かれているプログラムにはコメントは少ないけれども）．読者のプログラム本体に，コメントをつけてみよう．別の紙に書いてはいけない（紛失しないように）．プログラムに不慣れな読者は，プログラムの文はみないで，コメントを最後まで読んで，そのプログラムがどのように動くか，かなりよくわかるはずである．

プログラムを書きながらコメントをつけるべきである．さきにプログラムを全部書き，プログラムが動くようになってからコメントをつけるのは誤りである．ひとつには，とても大きなプログラムは決して完全には動作しない．そのために，コメントをつけることはできなくなるかもしれない．ときには，プログラムをデバッグするのがとても簡単でさきに解説をつける必要がないかもしれない．しかし，誰もが詳細におぼえている量には限界がある．プログラムにコメント付けをすることは，プログラムをデバッグする間中，プログラムがどう動くのかについてすべてを思いだす必要から読者を解放する．

別の考えでは，最後にコメントをつけられたプログラムは，適切にコメントをつけられていることがほとんどない．そして，よくコメントがつけられているプログラムほど維持しやすい．2年前に書いたプログラムを変更しなければならないとき，適切なコメントがつけられていたらとてもうれしいだろう．

コメントは，随筆の一部であるかのように，主語，述語，目的語を持ったよく整った文章であるべきだ．簡略語や文の断片を使用して，最小になるようにする方がよい．正しく几帳面にコメントをつけるべきである．いいかえると，英語を使用するルールをコメントでも有効に使用するべきである．そして，コメントをプログラムのような，読みにくいものにしてはいけない．

よいプログラマはよく，プログラムをまったく書かないうちに，コメントをさきに書く．つまり，文章でまずプログラム全体を書き，プログラムの動作の細部をすべて詳しく記述する．それから，コメントにコードを添える．処理をことばで記述することは，処理を分類するのに役立つ．考えつかなかった潜在的な問題があきらかになる．したがって，C言語で書く前に，文章で書いていることを書きおろすならば，しばしばよりよいプログラムになるだろう．

5.3.1　空白（white space）とインデント付け（indenting）

適切に分布された空白は，自由につけることができるもっとも有効なコメントのひとつである．たいていの平凡な書類にあるように，空白はプログラム中ではいつも句読点（,；等）の後に続くべきであり，できれば演算子（＋，－，＊等）を囲んでいるべきである．空行が段落の前にあるように，プログラムは，空行を前において，機能的に短くかためて配置されている．**while**，**for**，**do**，**switch**，**if** 文は，いつも空行が前にくる．else 文は，選考する if 文の一部だから，通常

は空行を前におかない．一般的に，空白なしで5，6行以上は続けない．

必ず，1行には1文以上は書かない．プログラムを次のように書くべきではない．

```
if(isdigit(*s))for(p=s,i=0;isdigit(*p);)i+=i*10+(*p++-'0');
```

（プログラムを読みにくくしたり，デバッグできないようにするつもりがなければ）この例からは，得ることは何もない．この行は，次のように書かれるべきだった．

```
if( isdigit(*s) )
{
        i = 0;
        for( p = s; isdigit(*p); p++ )
                i += (i * 10) + (*p - '0');
}
```

インデントをつけることは空白と関連した問題である．文の本体（body）は，少なくとも4つの空白のインデントを置いている．ここにいくつかの例がある．

```
while( condition )
        body( );

for( a; b>c; d++ )
{
        body_of_for( );
}

if( condition )
        action( );
else
        another_action( );
```

プログラマの中には，空白4つのインデントさえ渋々認めたり，ひとつか2つしかあけない人もいる．しかし，空白ひとつのインデントではプログラムは全然読みやすくならない．C言語でプログラムしているのである．BASIC —— 余分な空白はプログラムの実行を遅くする —— ではない．インデントをつけているためにプログラムの右端が書ききれないなら，そのときは，たぶん低いレベルの入れ子を独立したサブルーチンにするべきだろう．入れ子が多すぎるなら，間違いなく設計の欠陥の印である．プログラムが“ただ大きくなってしまっている”ときには起こりやすい．つまり，本当の問題はタブが大きすぎるのではなくて，サブルーチンが十分に考えられていないということである．

'}'，'{' は，もっとも外側のインデントのレベルであることに注意すべきである．これは，対応のない '{' と '}' をとても簡単に発見させる．つまり，

定規で線をひくことができる．かっこにインデントをつけているなら，かっこは囲んでいる部分に見失われやすい．同様に，読者が次のようにするならば，対応しないかっこをみつけにくい（斜めの線をひかなくてはならないし，'{' を見失いやすい）．

```
while( condition ){
        code( );
}
```

インデントをつけることはヌル文にも適用する．例えば

```
for( c = x;  c  ; c++ )
           ;
```

ここで，セミコロンはインデントをつけられて，次の行にひとつおかれている．for や while 文の終わりの不必要なセミコロンは，筆者がよくおかすエラーである．例えば

```
while( a < b );
{
        a++;
}
```

は “a が b より小さい間は何もしない．そして，それを処理したら a を増加させる” ということである．しかし，上のプログラムではそのループは決して終わらない．同様に

```
if(     条件      );
        do_something( );
```

は “条件が真ならば，何もしない．そして do-something( ) を呼ぶ”．次の行にセミコロンをおくことで，この種のエラーをみつけるために grep を使用することができる（1 章参照）．つまり，正規表現 while. *; は，セミコロンが後に続く語 while をすべての行で探す．for. (. *); は誤った終わり方をしている for 文を探す．

かっこのおき方のルールで問題のないひとつの例外は do/while 文が次のように書かれているときである．

```
do {
        code( );

} while(     条件      );
```

この構成は，たった今述べたバグ（誤っておかれたセミコロン）と while 文に正当なセミコロンが続く状況（その行に先行するかっこがあるのなら，そのときは最後のセミコロンはそこに属する）とを区別させる．

最後のひとつの問題は，同じページにあるサブルーチンの視覚的な区切りである．それらのサブルーチンがともに走るなら，それらを発見しにくい．したがって，各サブルーチン宣言の上に

```
/*------------------------------------------------*/
```

のようなダッシュをひいたコメント行をおくことはよい考えである．

サブルーチン本体の内側や目的（サブルーチンどうしを分ける）に反することに，このようなコメントをおくべきではない．サブルーチン内の主なブロックを分けるには2，3行の連続した空行を使うべきである．

5.3.2 コメントの書き方

Cでは，コメントを入れ子にできない．次のプログラムを考えてみる．

```
コメントはここから始まる
 |
 |
 |
 /* これはきちんと終わっていないコメントです．数行にまたがる
 比較的長いコメントになってしまった．

 while( a )
         a = foo( );


 /* これは別のコメントです．これはきちんと終わっている．しか
 し，コメントの終わりの文字がみつけにくい．*/
                                             |
                                             |
                                             |
                                         ここで終わり
```

このプログラムでは，while ループがコメントの一部になっている．この問題は終わりの記号を見やすくするようにコメントを書くことで小さくできる．2つの例がある．

```
/*   これはたぶんもっともよいコメントの書き方である．星印がついてい
 *   て，まとまったコメントをすぐみつけられる．このようなコメントは，
 *   それがみつけられるブロックと同じ入れ子レベルにいつも書かれるべ
 *   きである．
 */

/*   これは，コメントの始まり（open-comment）と     */
/*   コメントの終わり（close-comment）の記号がき    */
/*   ちんと縦に並ぶ場合だけはよい．                 */
```

コメントをでたらめにおいてはいけない．あるいはコメントの記号を右端，または左端にそろえて書くこと．これだと簡単で終わりの記号を忘れることがない．

5.3.3 コメントをおく場所

2，3のコメントは，読者のプログラムにも必ずあるだろう．ファイルの先頭にあるコメントは，ファイル名，そのファイルにあるサブルーチンやプログラムの動作内容の簡単な記述，外部から受け入れるすべてのオブジェクト（サブルーチンとグローバル変数）を与える．筆者は，図5·9に示される書式を用いている．

```
#include <stdio.h>                    /*  すべての#includeがここにくる    */

/*
 *       FILE.C  -  ファイルが行うことをここに簡潔に書く．必要ならば，
 *                  数行になってもよい．
 *
 *       外部オブジェクト：
 *
 *       foo( a, b )       -  foo( ) がすることをここに書く
 *       int a;            -  a がすることをここで説明する
 *       int b;            -  b がすることをここで説明する
 *
 *       bar( d, e )       -  bar( ) がすることをここに書く
 *       double d;         -  d がすることをここで説明する
 *       char   *e;        -  e がすることをここで説明する
 *
 *       int  Global_1;    -  Gloval 1 がすることをここに書く
 */
```

図 5・9 ファイルの先頭のコメント

各サブルーチンの先頭にある同様のコメントが，そのサブルーチンの動作内容とサブルーチンの動作レベルはどのくらいか，すべてのパラメータのすることを記述しているはずである．すべてのローカル変数の機能も，コメントに書かれている．図5·10がその例である．

```
void    foo( a, b )
int     a;                    /*  a がすることをここで説明する    */
int     b;                    /*  b がすることをここで説明する    */
{
        /*      このコメントで foo( ) がどのように動作して，何をする
         *      のかを説明する．
         */

        int     local1;               /*  local1 がすることを説明する   */
        char    *local2;              /*  local2 がすることを説明する   */

        /*  実行文がここから始まる．
         */
}
```

図 5・10 サブルーチンのコメント付け

プログラムを書き散らしてはいけない．プログラム文と交互にコメントがある

と，コメントなのかプログラム文なのか判読しにくくなる．それはまるで，2冊の本を1行ずつ交互に読もうとしているようである．コメントとプログラム文は，論理的なブロックにまとめられるべきである．プログラム文のブロックのまえに，そのプログラム文が，どのように動作するのかを記述したコメントのブロックをおく．その動作が複雑であれば，プログラム文に対応するコメントに番号をつけることができる（脚注のように）．図5·11がその例である．コメント（"/*1*/"等）は，while文の上の記述にある番号（（1）等）を参照する．

```
/*
 *        これは下の while ループ内で行われることを説明する複雑なコメント
 *        である．そのループは次のように動作する：
 *        (1)       do something の動作内容
 *        (2)       do something else の動作内容
 *        (3)       do another の動作内容
 */

 while( condition )
 {
          do_something( );          /*  1  */
          do_something( );

          do_something_else( );     /*  2  */
          do_something_else( );

          do_another( );            /*  3  */
          do_another( );
 }
```

図 5·11　実行文のところからコメントを参照する

5.3.4　明白なことを説明しないこと

コメントは，みてすぐにわからないことを教える．次のようなコメント

```
tea += 2;        /* tea に 2 をたす            */
elisa( )          /* サブルーチン elisa 開始    */
```

は役にたたないどころかむしろ悪い．これは，みただけではわからないことを何も教えないばかりか，不必要に冗長なページにしている．プログラムを読む人はだれもが言語を知っていて，演算子がどう動くか理解していると仮定していれば安全である．

5.3.5　変　数　名

変数名は，それ自体変数の機能を示すコメントである．変数名は何かを意味していなければならないし，変数はその機能と同様にそのスコープがわかるように

つけられているべきである．

マクロ名（♯define）は，すべて大文字で書くべきである．グローバル変数の最初の1字だけは，大文字にすべきである．ローカル変数はすべて小文字である．このようにすると名前をみることで変数のスコープがわかり，同じ名前のグローバル変数と，ローカル変数を互いに区別できる．変数名の中間の文字を大文字にすると，そうした理由を思いだすのが困難になる．2つのことばを分けるには下線（_）を使用すべきである．それは一貫して行う必要がある．

変数名は，その機能を表すべきである．さらに，できれば短縮名にしない（短縮は読みにくい）．ここでの問題は移植性である．初期のCコンパイラは，変数名の最初から8文字を識別のために必要とした．余分な文字は，もしあっても無視された．コンパイラは，長い変数名をサポートし始めている．しかし，8文字より長い名前を使用すると，さきのコンパイラには移植できない．短縮形にするときは，よく注意する必要がある．自分は理解している短縮形も，他の人にはわからなかったりする．

コンパイラが長い名前をサポートしていて，アセンブラがサポートしていないならば，マクロを使用してこの問題を回避することができる．例えば

```
#define     this_is_a_long_variable_name    vn1
this_is_a_long_variable_name = 6;
```

C言語のソース・プログラムでは長い変数名を使用できるが，アセンブラはそれを短くして等価なものを読ませる．けれども，この方法は注意を払う必要がある．プリプロセッサの中には，たとえコンパイラ自体が長い文字を受け入れても♯defineの最初の8文字しかみないものもある．プリプロセッサは，コンパイラの一部ではない．

5.4 グローバル変数とアクセス・ルーチン

どうしても必要なとき以外，グローバル変数は使わない方がよい．使用する場合は，グローバル変数はstaticで宣言し，アクセス・ルーチンを通して外部から更新するべきである．

ここでの問題は，リンカによって起きる．同じ名前の2つのグローバル変数が，それぞれ別にコンパイルされるモジュールに宣言されているならば，たいていのリンカはそれらを同じ変数だと思ってしまう．5年前に，ひとつのモジュールを

書いてライブラリに入れたとする．そして，それから偶然にもそのライブラリ・ルーチンにある名前と同じ名前のグローバル変数を使用したとする．その変数が，時々（そのライブラリが呼びだされるとき）不思議なことに，値がかわってしまうのがわかる．この状態は，発見されにくくかつ修正しにくい．

この問題の解決法は，すべてのグローバル変数をstaticで宣言することである．静的なグローバル変数は，それが宣言されているファイルの範囲に限定される．したがって，あるファイルに宣言されたサブルーチンは，別のファイルに宣言された静的なグローバル変数をアクセスすることはできない．2つの静的なグローバル変数が同じ名前であっても，リンカはその2つを別の変数として扱うだろう．

時々ではあるが，他のモジュールからグローバル変数を変更する必要があることがある．そして，このためにアクセス・ルーチンが使用される．アクセス・ルーチンは，静的グローバル変数を修正したり取りだしたりするサブルーチンである．アクセス・ルーチンは，それ自身はグローバルにアクセスできる．図5·12は，そのようなルーチンがどう使用されるかを示す．ルーチンsetvar( )とgetvar( )は，変数Global_varアクセスする．アクセス・ルーチンは，7章にある式解析で使用される．そこで，よりくわしく討議する．

図 5・12　アクセス・ルーチン

ファイルを機能的に構成しているならば（そのために同じファイルにあるすべてのサブルーチンは機能的に関連がある），ある特別なグローバル変数を使用す

る全サブルーチンは，たいていひとつのファイルに集められている．したがって，実際にはアクセス・ルーチンはあまり必要ではない．

5.5 移　　植　　性

Cの長所のひとつは移植性である．普通，マイクロプロセッサのタイプによって，プロセッサ専用のCコンパイラがある．しかし，C言語が，移植性がよいという事実は，必ずしも移植できるということを意味しない．うまく書かれたCプログラムは，モデムを通して他の計算機にダウンロードし，再コンパイルして，少しの労動で実行させることができる．一方，移植性を考えないで書かれたプログラムは，他のシステムで実行させるには何日もかかるだろう．

移植性を考えるうえでもっとも重要なことは，たぶん，コンパイラのI/Oライブラリである．UNIXコンパイラのI/Oライブラリは，C言語用に事実上標準のものを与えていて，多くのコンパイラがUNIX I/O関数の主なものをサポートしている．読者のコンパイラのライブラリがUNIXコンパチブルでなかったら，他のコンパイラを求めたほうがよいかもしれない（少なくとも他のシステムにプログラムを移動するつもりなら）．UNIX Programer's ManualのVolume I（参考文献参照）には，すべてのUNIX I/Oルーチンが記述されていて，自分のコンパイラの解説書と比較する基本としてたいへん役にたつ．

コンパイラの中には，C言語に役にたつ魅力的な拡張を加えているものがあるが，それは使用しないこと．同様に，コンパイラのメーカーが供給している標準でない（UNIXではない）サブルーチンは，そのソース・プログラムがついていなければ，そのようなサブルーチンは使用しないこと．

ハードウエアについては，何も想定していない．ハードウエアを操作しなければならないルーチンは，ひとつのファイルに集めるべきである．そうすれば，必要なとき簡単に変更できる．intが16ビット幅で，ポインタとintが同じ大きさであると仮定してはいけない．図5·13に，intのビット幅をバイト数でかえすサブルーチンを示す．

整数の大きさは，時々定数で示される．このため整数の大きさはいつも

```
0xfffe
```

よりも

```
~0x1
```

```
width( )
{
        /*      int のビット数をかえす
         */

        register int    i, j;

        for( j = ~0, i = 1;  j <<= 1 ;  i++ )
                ;

        return i;
}
```

図 5・13　int のけた数をみつける

の方がよい．

さきの式は，32 ビット幅の整数では動作しない．同じく，使用するならば

```
0x8000
```

の代わりに

```
~(( (unsigned)~0 ) >>1)
```

を使用するべきである（~0 は int だから，ここではキャストが必要である．それで，符号拡張は数がシフトしたとき行われるかもしれない．最上位ビットをクリアすることにはならないだろう）．移植性については，10 章でより深く討議する．

5.6　避けるべきこと

5.6.1　goto の乱用：非構造化プログラミング

本当に必要がなければ goto 文は使用すべきではない．goto それ自体，悪いところはない．多くの状況（いくつものレベルにいれ子になった while や for ループから抜けださなければならないような場合）では，goto を使用することが，もっとも簡潔で効率よくこの問題を解く方法である．C 言語は，ループを制御する方法を 3 つ（for，while，do）と，ループの制御の流れをかえたり終わらせたりする 4 つの方法（break，continue，return，exit( ) サブルーチン）を持っているから，goto はめったにプログラムの中で必要とされない．

goto の乱用は，C プログラムが goto と同じサブルーチン内にあるべきラベルを必要とする場合に制限される．goto を使用するための大まかな原則を次に示す．

（1） while，for で，break，continue，return，exit( ) が使用できるならば

```
main( )
{
        while( ... )
        {
                while( ... )
                {
                        while( ... )
                        {
                                if( an_unrecoverable_error )
                                        goto abort;
                        }
                }
        }

abort:
        clean_up( );
}
```

図 5・14 gotoの有効な使用法

goto は使わない．goto 以外のものを使うべきである．だからそれらを使用すべきである．

（2） goto と分岐しようとするラベルが，プログラム・リストの同じページにある場合のみ使う．そうすれば，制御の流れがひと目でわかる．ラベルは，インデントのレベル（図5・14参照）にあわせないで，もっとも左のカラムから書き始めてみやすくする．

（3） goto に対応するラベルは，必ずそのサブルーチンのもっとも外側のブロックにある文につながること．if 文から，else 節中にあるラベルに分岐するようなことは絶対にしないこと．

（4） パニックになるのを避けるために，goto 文の使用は厳密にすること．サブルーチン内のラベルはひとつだけであること．複数の goto 文からそのラベルに分岐することはあっても，複数のラベルには絶対に分岐すべきではない．

goto に関連する問題は，setjmp( ) と longjmp( ) サブルーチンの呼出しである．setjmp( ) の呼出しは，現在の計算機の状態をバッファに保存する．longjmp( ) の呼出しは setjmp( ) する前の状態にもどす．これらのルーチンは，プログラムの維持管理の悪夢を生みだす．スタック・フレームは破壊され，変数はよくわからない状態で発見される．さらに，制御の流れはプログラムをみても追跡できないようになってしまう．筆者は，これらのサブルーチンを使

用せずにすんだし，読者のプログラムが正しく構造化されていれば，読者も使用する必要はない．setjmp/longjmp は，始めからきちんと構造化されていないプログラムに起こるバグを修理するために，使い過ぎる．これらのサブルーチンを使うよりも，プログラムを書き始める前に計画を練るために時間をかける方がずっとよい．

5.6.2　♯define の誤使用

筆者には，“どんな言語でも FORTRAN プログラムを書くことができる”という友達がいる．不幸にも，彼は本当にそういっている．他の言語と同じように C 言語を考えようとしてはいけない．

```
#define begin    {
#define end      }
#define AND      &&
#define OR       ||
#define Stuff    stuff( )
```

下のようにできるからといって，上の define をしてはいけない．

```
while（条件　AND　条件　OR 条件）
begin
        Stuff;
end
```

読者が Pascal でプログラムを書きたいのなら，Pascal を使いなさい．C 言語を他の言語と同様にみなすことは，他の言語を知らない C プログラマが，読者のプログラムを管理するのをただ難しくするだけである．そのプログラムは必ずしも読みやすくはない．ただ，より親しみがあるようにみせているだけである（それは，たまたま読者が Pascal を知っていたからに違いない）．多くのプログラマは，知らない言語を，よく使っているものに似せようと努力する．初めてその言語を学ぶとき，このようなことをしがちである．自分ではとてもはやく C 言語の構文に慣れていくのがわかるだろうが，たぶん最初の頃の C プログラムのほとんどを Pascal もどきにしてしまうだろう．

関連した問題に

```
#define TRUE     1
#define FALSE    0

while( something == TRUE )
        stuff( );
```

がある．ここでの問題は，C は 0 でない値が真だということである．それで，た

とえsomethingが真でも，条件something ＝＝ TRUEは真にならないことがある．式がsomething！＝ FALSEならば大丈夫だが，プログラムに不必要なことばを加えている（!somethingでも同じであるから）．前の例のように，考えるよりもはやく適切なC言語の構文を使えるようになるだろう．

5.6.3　switch

ブロックの内側にcase文を並べないこと．これで正しいのだが，次のプログラムのような例はすすめられない．

```
switch( x )
{
case a:
        if( some_condition )
        {
                do_something( );
        }
        else
case b: {
                do_something_else( );
        }
}
```

同様に，caseはbreakがなければ次のcaseに抜けおちるから，switchの中のすべてのcaseは，break文か，故意にbreakを書いてないことがわかるコメントで終わりにするべきである．

```
switch( x )
{
case a:
        code( );
        /*  このまま case b にすすむ   */
case b:
        more_code( );
        break;
}
```

5.7　if/else

一般に，if節の次にわずかな空白も持たないでif/elseのブロックがあるようにif/else文を構成してみなさい．

```
if( condition )
        short_block( );
else
{
        /*
         *
         *
```

```
                                */

                                long_block( );

                                /*
                                 *
                                 *
                                 */
                        }
```

ここで，問題は読みやすさである．else の実行を制御する文を else と同じページにおこうとしている．さきに短い節をおくと，これがたいてい確実に行われる．

5.8　C のスタイル・シート

このセクションは，本章で討議してきた書式の全問題の概要である．プログラムの内容は問題にしない．しかし，むしろ C 言語でどのように一般的な構成が書かれるべきか，その例を簡潔にまとめたものを示す．読者のプログラムをこのように書くことを強くおすすめする．自分の書式をかえるのならば，変更した結果はより読み易さが増すのでかえるとよい．とにかく，しばらくの間それをやってみる必要がある．1 度よい書式を使ったら，そのときは絶対的なルールのようなものに思われる理由がわかるだろう．よい書式は本当に役にたつ．

```
よくない：      for(i=2+16;i<=MAXCH&&i>0;i+=2)foobar=i;

よい：          for(i=2+16; i<=MAXCH && i>0; i+=2)
                        foobar=i;

もっとよい：    for(i = 2 + 16;  i <= MAXCH  &&  i > 0; i += 2)
                {
                        foobar = i;
                }
```

```
==========================================================================

#include <stdio.h>

/*      FILENAME.C はファイル名．ここでこのモジュール内のルーチンを記述する．
 */

/*-------------------------------------------------------*/

extern  char     *foo();      /*  foo が宣言されているモジュールの名前        */
extern  unsigned bar();       /*  bar が宣言されているモジュールの名前        */
extern  int      Globalvar;   /*  Globalbar が宣言されているモジュールの名前  */

/*------------------------------------------------------
 *      #define と typedef はその名前はすべて大文字である．
 *      グローバル変数は最初の一文字だけが大文字である．
 *      ローカル変数はすべて小文字である．
 *      構造体や配列を初期化するときは，縦にそろえて書く．
 */
```

```
#define ARRAYSIZE        (1024 * 6)  /* 読みやすいので7224と書くよりこのほうが
                                      * よい．コンパイラがコンパイル時に式を評価
                                      * する．
                                      */

typedef struct _foo                  /*  この構造体の用途を説明する  */
{                                    /*                             */
        char            a;           /*  aの説明                    */
        int             b;           /*  bの説明                    */
        struct _foo     *ptr;        /*  ptrの説明                  */
}
FOO;

static int      Var     = 1;         /* varがどのように使用されるか説明する */
static char     *Var2   = 2;         /* var2がどのように使用されるか説明する*/


/*  Var3とVar4の説明をする．
 *  定義の横に余白がないのでコメントが書けない．
 */

static FOO      Var3    = { 'a',  "string",          &var4        };
static FOO      Var4    = { 'b',  "another string",  (FOO *)NIL   };

/*-----------------------------------------------------------*/

main( argc, argv )
int     argc;
char    **argv;
{
        /*  このコメントは大ざっぱにプログラム（またはサブルーチン）の説明をする．
         *  各ひき数の働きを説明し，かえり値とその意味を書く．
         */

        int             local_int;                /* local_intがすることの説明 */
        char            array[ARRAYSIZE], *ap;    /* array, apがすることの説明 */
        static int      var_2;                    /* var_2がすることの説明     */

        while( statement )
        {
                /*
                 *      whileループで起こることを説明する（みただけではわからな
                 *      ければ）
                 */

                body( );
                more_body( );
        }

        for( initializer = 0 ;  comparison( ) ; modify(initializer) )
        {
                for_body( arg1, arg2, arg3 );
                for_body( arg4, arg5, arg6 );
        }

        if( a )
        {
                statement( );
        }
        else if( b >= a  &&  d < b  )
        {
                statement( );
        }
        else
        {
                statement( );
        }
 }
```

```
/*-------------------------------------------------------*/

a_subroutine( a1 )
int     a1;
{
        /*      説明文
         */

        switch( a1 )
        {
        case CR:
                do_something;
                break;

        default:
                printf("a_subroutine( ): bad argument <%c> (0x%04x)\n",
                                                        a1, a1);
                break;
        }

        while(1)
        {
                if ( !do_forever)
                        break;
                else
                         do_something( );
        }

        do {
                --a;
                more_stuff( );

        } while ( a1 );
}

/*-------------------------------------------------------*/

char    *another_routine(b)
int     b;
{
        switch(b)
        {
        case 'a':       return ("a");
        case 'b':       return ("b");
        case 'c':       return ("c");
        case 'd':       return ("d");
        default :       return ("?");
        }
}
```

5.9 練　　習

5-1　右端もそろった英文テキスト（right-justifies text：英文の右端が縦に一直線上に並ぶように，1 行にできるだけ均等な間隔をあけて単語を配置してある）を作成するプログラムを設計しなさい．標準入力から入力し，標準出力へ書きだす．入力の最初の行は，出力するテキストのカラム数を含んでいる．入力には gets( ) を，出力には puts( ) を使うべきである．

ひとつのサブルーチンが，文の行ぞろえをするために呼びだされ，それは適当な文字配列の内容を変更する（別の配列にコピーしないこと）．

5-2　練習5-1で設計したプログラムを書きなさい．すべてのstubを示して，プログラム開発過程の各段階をすべて書きだしなさい．

6 ポインタ

本章は，C 言語において，もっとも誤解が多く，とても重要な部分であるポインタを詳しく検討する．10 倍もはやくなったというのは聞いたことがないが，ポインタは，プログラムの実行時間をかなり改善する（注 1）．実際，配列を使用して，ポインタでアクセスしないプログラムは，まったく C プログラムとはみなせず，C 言語で書いた FORTRAN プログラムだと思われるほど，ポインタは，C 言語の不可欠な部分である．

不幸なことに，たいていの C 言語の参考書ではポインタはまったく扱われていないか，いい加減に扱われている．本章では，たいていの C 言語の参考書に書かれているポインタについて扱う．そして，読者は他の教科書と並行して本章を読んでもかまわない．最初の 2, 3 セクションは，計算機内で表されるポインタについて記述する，メイルボックスのような形態は使わない．これらのセクションでは，読者が C 言語での宣言のしかたを知っていて，計算機のアドレッシングも理解していると想定している．次章 "高度なポインタ" では，ポインタの複雑な使用法を詳しく調べる．本章に書いてあることを完全に理解するまでは，次章に移るべきではない．

本章を最初に読むときには混乱するだろうが，詳細をつけ加えるために注釈を用いた．最初に読み通すときには，それは無視するとよい．構造体へのポインタは，章の終わりで述べるけれども，たいていの本で適切に扱われているので，深

注 1：ここでは，8085 上を走るプログラムについて述べる．構造体の 2 次元配列へのすべての直接参照は，ポインタ参照にかわった．速度の向上は，いつもこれほどめざましくはない．しかし，それはしばしば，3 つか 4 つの要因のうちのひとつではある．なぜ，そうなるのか，少しみていく．

く説明はしない．

6.1　簡単なポインタ

ポインタは，他の変数のアドレスを保持した変数である．この定義はおぼえておくべきである．ポインタは，他の変数の**アドレス**を保持した**変数**である．ときには，ポインタのようなアドレス自体を参照することもある．この定義をくわしく解析しよう．まず最初に，ポインタは変数である（定数に対して）．それについて特別なことはない．式中に使われるとき，変数名はその変数の内容に評価される．変数（または式）が，何に評価されるかみるよい方法は printf( ) 文を使うことである．例えば

```
printf("%d",x);
```

は x の内容をプリントする．変数 x は，文中で使用されるときはその内容に評価される．

C 言語では，オブジェクトを宣言することで，そのオブジェクトが使うメモリ領域を得る（注 2）．宣言

```
int     x, y;
```

はメモリに 2 つの整数の大きさのオブジェクト用の場所を割り当てる．つまり，コンパイラはこの 2 つの変数が使うメモリをどこかに確保する．メモリの確保はどのコンパイラも行う．x にあるものが何かわからないから

```
printf("%d",x);
```

が実行される時点で何がプリントされるか予想する方法はない，メモリ中にある x と y の絵を図 6･1 に示す．アドレスは簡単化して示している．これで参照するアドレスをわかりやすく表すことができる．int は 16 ビットであると仮定している．

x と y は初期化されていなかったから，ごみを含んでいる．次の式で x と y に代入することができる．

注 2：定義と宣言の違いは，しばしば C に関する本の中に見受けられる．厳密にいえば，定義はメモリを割り当て，一方，宣言はオブジェクトをどのように使うかコンパイラに知らせるだけである．変数の定義は，いつも宣言を含んでいる．しかし，オブジェクトにメモリを割り当てないでオブジェクトを宣言することができる（extern 文を用いる）．リンカがプログラムをいっしょにするとき，実際のオブジェクトを発見する．本章では，宣言ということばを宣言と定義，どちらも意味する広いほうの意味に使用する．

```
変数名：                    変数の最初のバイトの
                            アドレス：

        |             |        50
        |-------------|
    x:  |     ???     |        52
        |-------------|
    y:  |     ???     |        54
        |-------------|
        |             |        56
```

図 6・1 2つのint変数

```
x = 100;
y = x + 1;
```

今，xは100を含み，yは101を含んでいる．メモリを図でみると，次のようになっているだろう．

```
        |             |        50
        |-------------|
    x:  |     100     |        52
        |-------------|
    y:  |     101     |        54
        |-------------|
        |             |        56
```

文x ＝ 100は100を変数xに入れる（内容を"かえる"ことができる．そういう理由(わけ)で変数と呼ばれる）．文y ＝ x ＋ 1は変数xに含まれている数をフェッチして，1を加え，それからyに入れる．変数名が式中で使われるとき，その内容に評価されるということを意味する．xが文y ＝ xで使用されるとき，100に評価される．なぜなら，その数がxに含まれているからである．

さて，これをすべてポインタに当てはめてみる．次の宣言を考えてみる．

```
int     *ptr ;
```

この文は，ポインタの大きさのオブジェクトにメモリを割り当てる．ポインタは，アドレスを保持する変数であることを思い起こしてみる．したがって，ポインタの大きさは，計算機上のアドレスを保持するために必要なバイト数である（8080，Z 80，PDP-11では2バイト，VAXや68000では4バイト，等）．int xのように，文int ＊ptrは1つのポインタに対してメモリを割り当てる．この位置のメモリの内容は，まだ変数が初期化されていないので，定義されていない．他の図を書いてみよう：

```
      |             |    50
      |-------------|
  x:  |     100     |    52
      |-------------|
  y:  |     101     |    54
      |-------------|
ptr:  |     ???     |    56
      |-------------|
      |             |    58
```

ptrは整数をさし示すとコンパイラに教えたけれども，その整数は，ポインタといっしょにつくられるわけではない．もし，配列へのポインタを宣言してもポインタだけがつくられる．配列はつくらない，それで図ではポインタはどこもさしていない．概して，ポインタが何をさすかに関係なく，すべてのポインタは同じ大きさである（注3）．メモリ割り当てからみると，ポインタ変数がintへのポインタか，longへのポインタか，配列か構造体へのポインタであるかどうかは問題でない．しかし，後で理由はあきらかにするが，コンパイラはさし示されるオブジェクトの型を知る必要がある．

ポインタは変数である．ほとんどの部分で，ポインタは他の変数と同様に使うことができる（注4）．例えば，ptr ＝ 100は100をptrという変数に入れる（注5）．文

```
printf("%d", ptr);
```

は100をプリントする（注6）．けれども，ポインタは，他の変数のアドレスを保持する変数である．メモリ番地100には既知の変数はない．ptrへ100を代入したことは，間違いではないが，何の意味もない

変数のアドレスは＆演算子でみつけられる．＆xは変数xのアドレスを評価す

注 3：これはいつも真実であるとは限らない．例えば，8086のミディアム・モデルでは，関数への16ビットのポインタとデータへの32ビットのポインタを，またはその逆で使用できる．多くのCプログラムはすべてのポインタを同じ大きさであると想定している．しかし，8086ファミリの計算機に移植するときは注意することが必要である．

注 4：実際には，ポインタで処理できることには制限がある．次は規則である．
（1） 代入．ポインタを，他のポインタの値や＆演算子で得たアドレスに設定することができる．
（2） ポインタと整数のたし算とひき算（2つのポインタをひとつにすることはできない）．
（3） 2つのポインタの間のひき算（その結果は2つのポインタの間のオブジェクトの差になる）．ただし，どちらのポインタも同じ型のオブジェクトをさし示していなければならない．
（4） 他の演算（かけ算，シフト，等）はほとんどできない．どうしてもしなければならないのなら，キャストを使ってこの問題を防ぐことができる．

注 5：多くのコンパイラは，ptr ＝ (int＊)100；とすることを要求する．

る（x の内容ではない）．文

```
ptr = &x;
```

は変数 x のアドレスをポインタにいれる．x はメモリ番地 52 にあるから，ptr = &x は 52 を ptr に入れる．メモリは今，次のようになっている．

```
        |              |     50
        |--------------|
    x:  |     100      |     52
        |--------------|
    y:  |     101      |     54
        |--------------|
  ptr:  |      52      |     56
        |--------------|
        |              |     58
```

ptr = &x は，他の代入と同様に，ただの代入である．不思議はない．

ポインタを通して間接的に x を変更するためには，＊演算子が必要とされる．C 言語での単項演算子＊は，object-pointed-to（被演算数がさし示すオブジェクトを示す）演算子である．＊ptr は ptr がさし示すオブジェクトを意味する．さし示されるオブジェクトは，ポインタに含まれているアドレスにあるオブジェクトである．ptr は 52 を保持している．それは x のアドレスである．従って，ptr にさし示されるオブジェクトは，メモリ番地 52 にある変数 x の内容である．

式＊ptr＝5 は，5 を ptr にさし示される x に入れる．すなわち，x＝5 と同じことである．要約すると，ptr（アスタリスクがない）は変数 ptr の内容を評価し，＊ptr（アスタリスクつき）は ptr にさし示される変数の内容を評価する．その変数のアドレスは ptr に含まれている．文

```
printf("%d, %d", ptr, *ptr );
```

は

```
52, 5
```

とプリントする．

注 6：ここでは本当に printf( ) をだましている．スタックにポインタの大きさのオブジェクトをプッシュして，printf( ) にはそれが int の大きさのオブジェクトで，まるで数であるようにプリントさせている．これはデバッグの間は役にたつ．しかし，ポインタが int より大きい計算機では動かない（なぜならポインタの値が切り捨てられてしまうから）．検討中の ANSI 標準において，printf( ) への引数 %p は，ポインタの大きさのオブジェクトを数のようにプリントすることが定められている．しかし，多くのコンパイラはまだこれをサポートしない．ときには，printf ( ) に次のように，ポインタが実際は long であると告げることによって問題を防ぐことができる．

p rintf("%ld", ptr)

次の ptr の宣言

```
int  *ptr;
```

はポインタの大きさのオブジェクトにメモリを割り当てる．しかし，そのオブジェクトは初期化されていない．したがって，ポインタがつくられたときは，あるオブジェクトのアドレスではなくでたらめな数を含んでいる．そのうえ，コンパイラは ptr が実際に有効な値を含んでいるかどうかわからない．最初に ptr の初期化（ptr ＝ &x か同じような文）をしないで＊ptr ＝ 5 としたら，どこかでたらめな場所（ptr に含まれているアドレス）に 5 を入れるだろう．もし ptr が 0 を含んでいたら，コマンド＊ptr ＝ 5 はメモリ番地 0 に 5 を入れ，そこにあったものを壊してしまうだろう．初期化していないポインタを使用することはよくある，危険なエラーである．メモリにあるものを何でも，プログラム自体も含めて完全に破壊することができる（注7）．MS-DOS では，ハードディスク上のディレクトリを破壊したり，オペレーティング・システム自体に上書きすることさえできる（注8）．このような災難を避けるためには，**ポインタを使用する前に必ず初期化するべきである**．

6.2　ポインタと配列名

配列についてだが，C 言語には一般の配列のようなものはない．著者がいいたいことは以上である．あるのは，すべて同じ型を持つひと固まりのオブジェクトに対して，メモリを割り当てる方法である．そして，このオブジェクトの固まりを "配列" と呼ぶことができる．しかし，この "配列" は，C 言語の型にはない．配列の要素は，ひとつのオブジェクトのように操作することができる．しかし，配列全体をひとつとして操作することはできない．つまり，実際に配列が型とし

注 7：多くの大きなメインフレームは，ポインタをデータに上書きさせる．しかし，それらはプログラムとは分けている．例えば，UNIX では制御できないポインタが命令コード上に書き込もうとしたら，"segmentation violation：core dumped" のエラー・メッセージを出すだろう．一方，多くのマイクロコンピュータは，このようなチェックは何もしない．"core" はプログラムが終了したとき格納されるコンピュータのメモリ・イメージを含んだファイルである．それは大きなファイルであまり役には立たない．rm core コマンドで消去することができる．

注 8：ディスクのディレクトリ構造の一部である FAT がメモリに格納され，ファイルが閉じるときにディスクに書きだされる．ラージ・モデルのプログラムのポインタか，またはスモール・モデルの far ポインタは FAT に上書きして，ログイン時のディスクのディレクトリを壊すことができる．

て存在するならば，ひとつずつ要素をコピーしなくても，等号で配列全体を別の配列に代入することができる．配列を値でサブルーチンに渡すこともできる．配列のサイズより大きな添字で配列の要素をさし示すことはできない．等々（注9）．配列の概念（オブジェクトの固まり，要素がメモリ内で連続していてすべて同じ型である）は存在する．しかし，配列は配列として操作できない．ポインタを使って，ひとつずつオブジェクトを操作しなければならない（しばらくの間このポインタの使用について述べる（注10））．

宣言

```
int      array[5];
```

は5つの整数にメモリを割り当てる（まだ初期化されていないので，このときはごみを含んでいる）．それらは必ずメモリ内で連続している．しかし，それらをひとつの配列としてではなく，5つの別個のオブジェクトだとみなすべきである．割り当て後のメモリを図6·2に示す．

5つの整数を割り当てているけれども，これらの整数は同じ名前ではない．どのようにそれらをアクセスすることができるだろうか？ C言語では，配列名はいつも配列の最初の要素へのポインタとして評価される．つまり，配列名は，配

```
          |             |     50
          |-------------|
      x:  |      5      |     52
          |-------------|
      y:  |     101     |     54
          |-------------|
    ptr:  |      52     |     56
          |-------------|
  array:  |             |     58
          |- - - - - - -|
          |             |     60
          |- - - - - - -|
          |             |     62
          |- - - - - - -|
          |             |     64
          |- - - - - - -|
          |             |     66
          |-------------|
          |             |
```

図 6·2 メモリに配列を加える

注 9：ポインタにかっこを使うことはどうしてもできない．

注 10：そのポインタは，配列名から派生した暗黙のポインタであるかもしれない．いずれにしてもポインタである．

列の最初のオブジェクトをさし示すように，初期化されたポインタ変数であるかのように使用される．しかし，配列名は定数である．ポインタのように操作されることはできない（注11）．しばらくの間これについて述べる．

配列の宣言と配列名を使用する方法を混同するべきではない．配列名がポインタのように使用されているけれども，配列とポインタはまったく違う．配列は，型の集合体（aggregate type）—すべて同じ型を持つ連続した変数の集合—である．ポインタは単独で，ポインタの大きさの変数である．宣言 int array[5] は，5つの整数にメモリを割り当てる．それには10バイト必要とする．宣言 int ＊ptr はひとつのポインタにメモリを割り当て，メモリに2バイトを必要とする．しかし，ポインタがアクセスするオブジェクトにはメモリを割り当てない．

本当は

```
char    buf[128];
gets(buf);              /* 正しい  */
```

としたいのに

```
char    *p;
gets(p);                /* 誤り    */
```

とするのはよく犯すエラーである（注12）．サブルーチン gets(　) は，引数として，標準入力から得る文字を入れるメモリ・ブロックのアドレスを期待している．配列名は，配列の最初の要素のアドレスに評価されるから，呼び出し gets (buf) は期待通りに動作する．buf は，128バイトのブロックの最初のバイトのアドレスに評価される．一方，p はひとつのポインタ・サイズの変数である．アスタリスクなしでは，p は変数 p の内容に評価される．呼出し gets(p) は，gets (　) に変数 p の内容を渡す．それは文字を入れるメモリ・ブロックのアドレスだと思われている．しかし，p はごみを含んでいる．何かをさし示すように p は初期化されていない．p にはメモリを割り当てられたとき，たまたま，そこにあったでたらめな数が含まれている．したがって，gets(　) はメモリのでたらめな位置に文字を入れ始める．もし本当に gets(　) に p を使いたいなら，次のようにするべきである．

注 11：アセンブリ言語では配列名は，配列の最初のセルに関係するラベルである．一方，ポインタは，メモリの他の場所のアドレスを含むメモリ中のセルである．つまり，配列名はラベルであり，ポインタ変数はラベルではない．

注 12：筆者はいままでに，C 言語を教えることを目的とする産業レベルの教科書でこのエラーをみた．

```
char    buf[128];
char    *p;

p = buf;
gets(p);
```

しかし，gets(buf) で十分なのに，わざわざ buf の最初の要素のアドレスをコピーする必要があるだろうか．

ここで問題になるのは，多くのコンパイラのマニュアルに書かれている gets( ) の使用方法である．次のように gets( ) の呼出し文が書かれている．

```
gets( str )
char *str;
```

表現は，このサブルーチンが引数として，文字へのポインタを必要とすることを意味する．Pascal のような型に厳しい言語に慣れているプログラマは，gets( ) に文字へのポインタとして宣言された変数を渡さなければならないと思いがちである．しかしこの考えは誤りである．マニュアルに，gets( ) が char へのポインタを期待していると書いてあるときは，型が char である変数のアドレスを期待しているということである（ポインタがアドレスを保持している変数であり，かつ，ポインタの変数名はその内容を評価することを思い出すとよい）．いくつかの方法でアドレスを得ることができる．もっとも簡単な方法は，配列の名前（かっこはいらない）を使うことである．char の配列をさし示すように初期化されたポインタの内容を gets( ) に渡すこともできる．

gets( ) は，その引数でさし示された文字に，いくつか char の大きさのオブジェクトが続く．つまり，gets( ) に渡されたポインタは，配列へのポインタであると想定している．しかし，配列名はその配列の最初の要素のアドレスに評価される．重要なことは，gets( ) にはアドレスだけが渡され，そのアドレスが配列のアドレスであるか，ひとつの char のアドレスなのか決定する方法がないということである．いいかえれば，配列へのポインタは同時にその配列の大きさの情報を持たない．サブルーチンは配列の大きさがわからないから，その配列は十分に大きいと仮定しているだけである．この処理はプログラマの責任である．

配列名に話をもどす．配列名は，配列の最初のセルをさし示すよう初期化されたポインタのように使うことができる．つまり，配列名は，その最初の要素をさし示すために初期化されたポインタと同様に，最初の要素のアドレスに評価される．例を使うと，文

```
gets( str )        /* サブルーチンgetsの定義：引数strはgets( )が文字を入れ   */
char  *str;        /* る配列の最初の要素のアドレスである.                    */
{                                                                            */
}

main( )
{
    char array[10];
    char c, *cp ;

    gets( array );     /* 正しい．"array" の最初の要素のアドレスを渡  */
                       /* す.                                         */

    cp = array;
    gets( cp );        /* cpが初期化されているときだけ正しい．文字配 */
                       /* 列のアドレスを保持しているに違いない，cpの */
                       /* 内容を渡す.                                 */

    gets( &c );        /* 間違い．構文は正しくて，エラー・メッセージ */
                       /* はプリントされない．文字のブロックの最初の */
                       /* アドレスよりもひとつの文字のアドレスをgets */
                       /* ( )に渡す.                                  */
}
```

図 6・3　ポインタ引数を持つサブルーチンの呼出し

```
printf("%d", array);
```

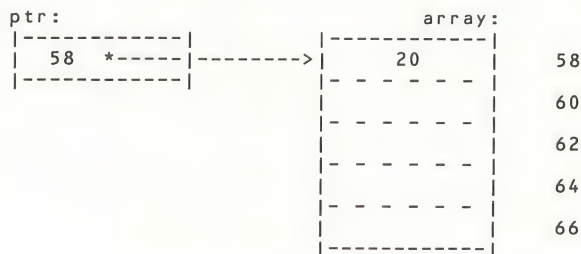
は58をプリントする（arrayの最初の要素のアドレス；printf( )に%sでなく%dを与えたことに注意せよ）．配列名は，いつもポインタであるかのように扱われる．宣言int array[5]であれば，式*array = 20はC言語では完全に正しい．コンパイラは，配列名をその配列の最初のセル（アドレス58にある）をさし示すように初期化されたポインタのように使用する．つまり，58を含んでいるポインタとしてarrayを使う．式*array = 20は20をメモリ番地58に入れる．

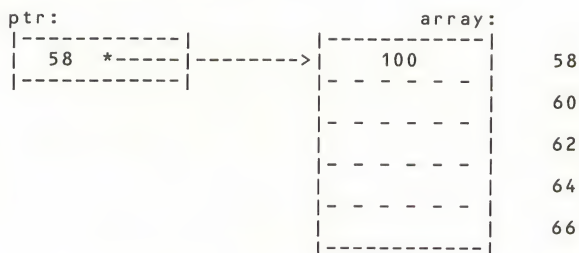
ポインタ変数ptrを使用する前に初期化していれば，ptrを配列への書き込みに使うこともできる．ptrは次の2つの方法のいずれかで初期化することができる．

```
ptr = &array[0];
ptr = array;
```

最初の文は簡単にわかる．&はアドレスを示す演算子である．式&array[0]は"array[0]のアドレス"，またはarrayの最初の要素のアドレス，すなわち58に変換される．配列名は，配列の最初の要素のアドレスに評価されるから，2番目の形式の代入文を使うこともできる．この2番目の初期化も，arrayの最初の

要素のアドレスを ptr に入れる．ptr は今 58 を含んでいる．すなわち，array の最初の要素をさし示している．その関係は次のように書くことができる．

```
ptr:                                  array:
|-------------|             |-------------|
|  58   *-----|---------->|      20      |      58
|-------------|             |- - - - - - - |
                             |              |      60
                             |- - - - - - - |
                             |              |      62
                             |- - - - - - - |
                             |              |      64
                             |- - - - - - - |
                             |              |      66
                             |-------------|
```

矢印は，ポインタとそれがさし示すオブジェクトとの関係を示す．ptr を初期化してあれば，それを使うことができる．式 *ptr = 100; はポインタ ptr に含まれているアドレスにあるオブジェクトに 100 を代入する．ptr は 58 を含んでいるから，100 がメモリ番地 58 に移動する．その番地は array の最初のセルである．メモリは今次のようになっている．

```
ptr:                                  array:
|-------------|             |-------------|
|  58   *-----|---------->|     100      |      58
|-------------|             |- - - - - - - |
                             |              |      60
                             |- - - - - - - |
                             |              |      62
                             |- - - - - - - |
                             |              |      64
                             |- - - - - - - |
                             |              |      66
                             |-------------|
```

通常のポインタ構文を使用して，さし示されているオブジェクトを更新することができる．例えば，*ptr = *ptr + 1 [または (*ptr)++] で ptr にさし示されているオブジェクトに 1 を加える (100 から 101 にかえる) ことができる．

6.3 ポインタの計算

ポインタと，ポインタとして使用される配列名には大きな違いがある．ポインタは変数である．変数の場合は，変数の内容を変更することができる．一方，配列名はポインタのように使われるけれども，変数ではない．定数である．ptr++ とすることができる．しかし，array++ とはできない (注 13)．

文 ptr = ptr + 1 (または ptr++) はポインタ自体に 1 を加える (x++ が x

に1を加えるように）．さし示されているオブジェクトは更新されない．ポインタだけが更新される．しかしポインタに対しては，1がひとつのオブジェクトを意味するということを間違えやすい．つまり，実際にはポインタの内容に1を加えてはいない．さし示されているオブジェクトの大きさが，ポインタに加えられている．ptrはintへのポインタだから（宣言int ＊ptrで定義されている）（注14），ptrには2が加えられる（intの大きさは2バイトだから）．文ptr＋＋はptrの内容を58から60にかえる．そしてptrは，arrayの2番目の要素をさし示す．今＊ptr ＝ 10とすれば，アドレス60のセルに10を入れる．メモリは次のようになっている．

```
ptr:                            array:
|------------|              |------------|
|  60   *----|---+          |    667     |    58
|------------|   |          |- - - - - - |
                 +---->     |    10      |    60
                            |- - - - - - |
                            |            |    62
                            |- - - - - - |
                            |            |    64
                            |- - - - - - |
                            |            |    66
                            |------------|
```

ポインタの計算は，すべての整数が式中で使われる前に，さし示されているオブジェクトの大きさをかけられているところが，通常の計算とは異なる（注15）．式

```
ptr += 3;
```

注 13：前の注釈で配列名がラベルであることを述べた．それを調べる別の方法がある．最初の要素のアドレスは，シンボル・テーブルに格納されているから配列名は定数である．つまり，最初の要素のアドレスを保持している変数はない．むしろ，コンパイラは，シンボル・テーブルにあるアドレスをおぼえていて，配列名が使用されるごとに即値命令（または，auto配列のフレーム・ポインタへの参照）を生成する．シンボル・テーブルは実行時には存在しないから，シンボル・テーブルの一部であるアドレスを変更することはできない．

注 14：ポインタを増加するとき，いくつポインタに加えるかコンパイラに知らせるのがポインタの宣言である．このことに気づくことは重要である．ポインタを増加させるとき，ポインタが実際に何をさし示しているかに関係なく，ポインタにsizeof(int）を加えるだろう．intへのポインタとして宣言された変数が，本当にintをさし示しているのかどうかコンパイラは関知しない．printf（9章でよく調べる）は，宣言されたオブジェクトの型以外のものをさしている，ポインタのよい例がある．

注 15：実際には，どのような整数型でも行う．式か，またはlong intを使うことができる．ただし，すべてのコンパイラがlong intを受け付けるとは限らない．例えば，p＋(x＊4）は，pがポインタで，(x＊4）がintに評価されるならば，正しい．

はptrの内容に［3 ＊ sizeof(int)］を加える（ptrにアドレス66をさし示させる）．式

```
ptr -= 3;
```

はptrをもとさしていたところをさすようにもどす．これらの式は，どちらもポインタを計算する．それで，ptrを減少させて，同時にさし示しているオブジェクトを更新するために

```
*( ptr -= 3 ) = 'x';
```

とすることもできる．

2つのポインタのひき算をすることもできる．ひき算の結果は，2つのポインタ間の差をオブジェクトの数（バイト数ではない）で表す．例えば，4番目の要素（array[3]）をさすポインタから，最初の要素（array[0]）をさすポインタをひくと整数3になる．ひき算の結果はポインタではなく整数になる．2つのポインタは，同じ型のオブジェクトをさしていなければならない．ポインタはほかのすべての算術演算ができない．

ポインタ計算とさし示されているオブジェクトへのアクセスは，ひとつの式に組み合わせることができる．例えば，＊と＋＋か－－をひとつの式に組み入れることができる．これを行うときは，演算子の優先順位に気をつけなければいけない．いくつか例をみてみよう．

```
x = *ptr++;
```

は一度に2つのことを行う．増加させられるものがあり，同時に，xに入れられるものがある．増加させられるものは何か？　ポインタ自体か？　それともさし示されているオブジェクトか？　この問題は，式を十分にかっこで囲めばわかる．Appendix Aの優先順位の図をみると，＊と＋＋演算子は同じ優先順位であることがわかる．しかし，右から左に関係づけられるから，式には次のようにかっこがつけられる．

```
x = *( ptr++ );
```

＊は式ptr＋＋全体に及ぶ（いい換えれば，＋＋の方が＊より強く名前に結びつけられている）．ポインタ自体が増加される．次に，変数名に＋＋が続くからこれは後置きのインクリメントであることに注意する必要がある．ポインタを使用した後で，それを増加する．次の質問は”式は何に評価されるか”ということである．アスタリスクの番である．式は，ptrにさし示されているオブジェクトの

内容（増加する前の）に評価される．それで，ptr にさし示されているオブジェクトを x にコピーする．それから，ptr を増加させる．

さて，次の式を考えてみよう．

```
x = *++ptr;
```

この式は，＋＋が変数名の前にあるから，ptr がさし示すオブジェクトがフェッチされる前に ptr が増加する以外は，先例と同様に動く．みえないかっこは

```
x = *( ++ptr );
```

である．

```
x = ++*ptr;
```

はどうだろうか．＋＋と＊の位置が交替している．この動作はさきほどとはかなり違う．式のかっこは

```
x = ++( *ptr );
```

である．＋＋はさし示されているオブジェクトに影響する．さし示されているオブジェクトが増加され，さし示されたオブジェクトの増加した値が x にコピーされる．ポインタ自体はかわらない．後置きインクリメントを使った等価な式は

```
x = ( *ptr )++ ;
```

である．これにはかっこが必要である．

今まで述べたすべての式は，さし示されているオブジェクトに評価される．いいかえれば，コンパイラにさし示されているオブジェクトをフェッチして，それを x に入れるよう教えている．x ＝が必要である．コンパイラにオブジェクトをフェッチして，それをどこにも入れないように教えることはできない．文

```
*p;
```

だけが行にあっても，文

```
x;
```

だけがあるのと同様意味がない．このような省略は "lvalue required" エラー・メッセージを生成する（10 章で詳述する）．

6.4　角かっこ（[ ]）表記（Square Bracket Notation）

ポインタ計算のために，実際にポインタを変更する必要はない．例えば，ptr が配列をさしているとする．次の式でさし示されているセルから，オフセット 3 のところにあるセルを変更することができる．

```
*(ptr + 3) = 5;
```

この式はコンパイラには，”ptr の内容に 3 ＊ sizeof(int）を加えたアドレスにあるメモリ・セルに 5 を代入する”と解釈できる（注 16）．いいかえれば，さし示されているオブジェクトは次のアドレスにある．

ptr の内容 + (3 * sizeof(オブジェクト)).

ptr 自体は変更されない．

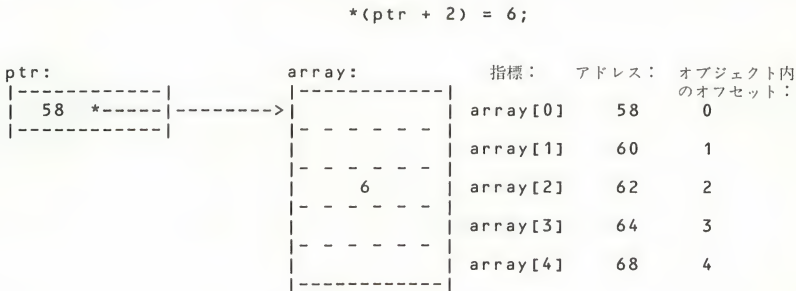



図 6・4 配列へのポインタ

図 6・4 に示された状況で，具体的な例をみてみよう．次の式で array の 3 番目の要素を変更できる．

```
*(ptr + 2) = 6;
```

(3 番目の要素は，その配列のベースからオフセット 2 のところにある.)

コンパイラは，ptr の内容（数 58）を取りだして，4（2 ＊ sizeof(int)）を加える．その結果は 62 になる．それから，6 をさし示されているオブジェクト（62 にあるオブジェクト）に移動する．ptr 自体は変更されない（たし算はその場かぎりの変数で行われる）．

配列名は実際，配列の最初の要素をさし示すために初期化されたポインタのよ

注 16：つまり，さし示されているオブジェクトの大きさの 3 倍である．

```
x + 1 = y;
```

ができないように

```
ptr + 1 = y;
```

または

```
*ptr + 1 = y;
```

とはできない

うに使用することができる．今は，ptr も配列の最初の要素をさし示しているから，次の式で同じ動作を行うことができる．

```
*(array + 2) = 6;
```

名前array は前の例でptr が扱われたのと同じに扱われる．

これがすべて，なんとか慣れ始める．実際に，等式

```
*(array + 2)    ==    array[2]
```

はすべての配列に対して維持される．[ ] 表記は単にポインタ演算の簡略表現である．この等式を一般化したものを次式に示す．

```
*(array + i)    ==    array[i]
```

配列は存在しないと前述した．コンパイラはポインタについてだけ知っている．配列名は，[ ] が続いても続かなくても，いつも配列の最初の要素に評価される．[ ] 表記は，コンパイラに次のことを指令する．

（1） [ ] の中にある数に，配列要素の大きさ（さし示されているオブジェクトの大きさ）をかける．
（2） 配列名で示されているポインタの内容に，オフセットとして（1）の結果を加える（配列名は，配列の最初の要素へのポインタに評価されるから）．
（3） 計算されたアドレスにあるオブジェクトを取りだす．

C 言語で演算子を操作するその方法は規則正しいから，[ ] 表記がポインタにも適用されることがわかる．つまり，次の式は ptr がどんなポインタのときにも維持される．

```
*(ptr + i)   ==   ptr[i]
```

配列名はポインタに評価されるから，この式は配列を操作するときに現れる．この等式の i ＝＝ 0 の場合は興味深い．

```
*(ptr + 0)  ==  *ptr  ==  ptr[0]
```

あるいは，これを言葉に置き換えると，ポインタ名の右に [0] があるときはいつも，[0] はポインタ名の左にある＊と置き換えることができる（注 17）．

[ ] の左にある名前は，配列名である必要はない．ポインタ名でもよい．配列名はポインタであるから，式

注 17：厳密にいえば，多次元配列（次章で述べる）に関しては当てはまらない．実際は次のようになる．

(ptr[x][y]...[z])[0] ＝＝ ＊(ptr[x][y]...[z])

```
int   array[5], *ptr;
ptr = array;
```

が与えられていれば，次のすべての式は同じことをする．

```
*(array + 3) = 5;
*(ptr   + 3) = 5;
array[3]     = 5;
ptr[3]       = 5;
```

ptr は変数であるから，次のようにも書ける．

```
ptr = array;
ptr += 3;
*ptr = 5;
```

ここでptr自体は変更される．しかし，array は定数であり，変数ではないから，array ＋＝ 3；とはできない．

これらすべての文が同じであれば，どうして［ ］表記の代わりにポインタを使用するのか？　答えは効率を考えることにある．x ＝ p[i＋＋] が実行されているとき，コンパイラに実際に命令していることを考える必要がある．

（１） i の内容を取りだす．
（２） さし示されているオブジェクトの大きさを，この内容にかける．その結果をそのときかぎりの変数に格納する．
（３） それから，i を増加してもとにもどす．
（４） p の内容を取りだす．
（５）（２）のかけ算の結果を（４）の結果に加える．
（６） 最後に，今計算されたアドレスにあるセルの内容を取りだしてx に入れる．

これには3つのフェッチ（取りだすこと）とひとつのかけ算がある．ここで，式 x ＝ ＊p＋＋ を考えよう．この式は同じ動作を行う（注18）．この2番目の式は，次のことを行う命令コードをコンパイラに生成させる．

（１） p の内容を取りだす．
（２） そのアドレスにあるオブジェクトを取りだして x に入れる．
（３） p にさし示しているオブジェクトの大きさを加える．
（４） たし算の結果を p に格納する．

ここには，フェッチが2つだけでかけ算はない（注19）．そのかけ算が，非効

注 18：これは，p が正しい配列要素をさし示すように初期化されているかぎり真実である．もちろん，初期化自体は，一度は行わなければならないというだけで，さきに述べた処理と同様に効率的ではない．

注 19：より正確にいうと，かけ算もやはりある．しかし，実行時ではなくコンパイル時に行われる．

率になる主な原因である．2章でみたように，かけ算は費用のかかる動作である．よってそれを使わなくてもよいようにするべきである．これは，ハードウェアでかけ算をサポートしていない8085のような計算機を使うときには特に重要な問題となる．また，たいていの計算機のハードウェアのかけ算は，ハードウエアのたし算の何倍も時間がかかる．もうひとつの非効率の要因は，オブジェクトの実際の大きさである．2のかけ算はただのシフトでよいが，6のかけ算（構造体や配列へのポインタで使う）は違う．一方で，多くの計算機は，2のべき乗のかけ算さえ，シフトを使わずにかけ算のサブルーチンを呼びだす．

オーバー・ヘッドの点から，p[i] と*(p + i) に差はないことに気がつくべきである．どちらの場合も実行時にかけ算がある．しかし，i が定数－ p[1] か*(p + 1) －であれば，そのときは実行時ではなく，コンパイラがコンパイル時にかけ算をすることができる．定数を使用する配列のアクセスは，変数を使用する配列のアクセスよりも速い．かっこと定数を使うアクセスは合理的な動作である．配列が static で宣言されていれば，必要なセルの実際のアドレスはコンパイル時に決定されてしまう．

同様に，配列をランダムにアクセスする必要があるならば，かっこを使用するべきである．その方が読みやすい．しかし，配列をでたらめにアクセスする必要はほとんどない．たいていは，規則正しいやり方で配列を操作する．列単位か，行単位か対角線に，などのやり方で．その場合は，ポインタがはやいと思われる．

6.5　文字配列；初期化

たいていの変数はそれらを宣言するとき，宣言に続く等号と初期値で初期化することができる．例えば

```
int     x = 5;
```

は変数 x にメモリを割り当て，同時に x を5に初期化する．固定したアドレスにある配列（グローバルか予約語 static を明示してあるもの）も同様に初期化できる．

```
static int      array[5] = { 5, 4, 3, 2, 1 };
```

ここで，array[0] は5に，array[1] は4に，などのように初期化される．実際は，配列を初期化するときは特に配列の大きさを規定する必要はない．例えば

```
static int      array[ ] = { 4, 5, 6 };
```

は3つの要素を持つintの配列をつくる．最初の要素は4に，2番目は5に，3番目は6に初期化される．次の式で，配列にある要素の数を決定することができる．

```
sizeof(array) / sizeof(array[0])
```

これは，バイト数で表した配列全体の大きさをバイト数で表したひとつの要素の大きさで割っている（注20）．

C言語では，暗黙の，宣言された配列：リテラル文字列がある．2重引用符（"）で囲まれたASCII文字列は，そのような文字列を形成する．それらの文字は，メモリの連続したバイトに入れられ，ASCII NULL（'\0' 数0）が最後につけ加えられる．式全体は，最初の文字が入れられているメモリへのポインタに評価される．例えば次の文

```
printf("Hello");
```

があるとする．コンパイラは，5つの文字（'H'，'e'，'l'，'l'，'o'）をプログラマにはみえない5つの連続したメモリ番地（そのアドレスをみつける簡単な方法はない）に入れる．それから0（数値0x30を持つASCII文字の '0' ではない）を6番目の番地に入れて，最初の文字のアドレスをprintf( )に渡す（図6・5をみよ）（注21）．

preintf("Hello");
は次の暗黙の配列を生成させる：

```
     |------------|
50:  |    'H'     |
     |------------|
51:  |    'e'     |
     |------------|
52:  |    'l'     |
     |------------|
53:  |    'l'     |
     |------------|
54:  |    'o'     |
     |------------|
55:  |   '\0'     |
     |------------|
```

そして数50をprintf( )に渡す．

図6・5 リテラル文字列

注20：これはしばしばマクロにされる．

```
#define array_size(x) (sizeof(x)/sizeof(*x))
```

注21：文字列" "は，'\0'を含むcharの大きさのメモリ・セルへのポインタとに評価される．

2重引用符は，文字へのポインタに評価されるから，文字へのポインタを初期化するときに，それを使うことができる．例えば

```
char    *msg = "Hello" ;
```

は2つのことをする．最初は，文字列 "Hello" に6バイトを割り当てて，それらを初期化する．それからmsgというポインタの大きさのオブジェクトにメモリの割り当てをして，'H' をさし示すようにmsgを初期化する．この状態を図6・6に示す．

```
msg:
|-----------|               |-----------|
| 60   *----|-------->|     'H'     | 60
|-----------|               |-----------|
                            |     'e'     | 61
                            |-----------|
                            |     'l'     | 62
                            |-----------|
                            |     'l'     | 63
                            |-----------|
                            |     'o'     | 64
                            |-----------|
                            |    '\0'     | 65
                            |-----------|
```

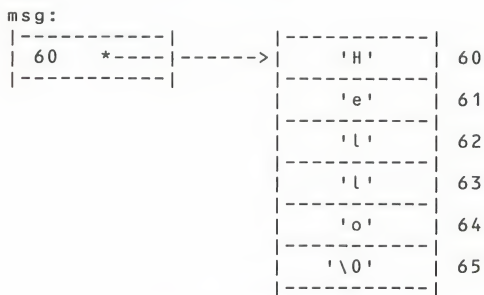



図 6・6 暗黙の文字列

ここで注意することが2, 3ある．まず，この配列は通常の意味の配列ではない．つまり，この配列には名前がない．そのため，直接アクセスすることはできない．それに達するにはポインタを使う．次に，msgは変数であるが配列ではない．配列をさし示すように初期化されている．しかし，msgはひとつで，ポインタの大きさのオブジェクトである．msgはポインタであるから，配列への他のポインタと同様に，配列をアクセスすることができる．例えば

```
x = msg[2];
```

はASCII 'l' (108) をxに代入する．同様に，*msgは文字 'H' に評価される．msgは変数だから，変更することができる．つまり，msg++とすることもできる．この変更はmsgが配列名だったら許されない．もちろん，msgがどこか他をさすように変更すれば，文字列は配列のように格納されているけれども名前を持たないから，再びその文字列をさし示す方法はない（注22）．

問題が混乱するけれども，次のように文字配列を初期化することもまたできる．

```
char    a[128] = "foo";
```

この宣言は128バイトの配列をつくるが，ポインタは生成しない．この配列の最初の4つの要素は文字 'f', 'o', 'o', と '\0' に初期化されている．次のようにすることもできる．

```
char    a[128] = {'f', 'o', 'o', '\0'};
```

これはかん違いは少ないが不便である．コンパイラの中には，暗黙の文字列に128文字の制限をおくものもあるので，この2番目の形の初期化をとても長い文字列で使用しなければならないかもしれない（注23）．

printf( ) のサポート・ルーチンの中に，引用符で囲まれた文字列を使う面白い応用がある．次の文を考えてみる．

```
putchar(  "0123456789abcdef"[ x & 0xf ]  );
```

引用符で囲まれたすべての文字列は，その文字列の最初の文字へのポインタに評価される．また，[ ] 表記はどんなポインタにも，たとえ暗黙のポインタでも，適用できる．したがって，x が0から15の範囲の数（& 0xf がそれを保証している）であれば，x はその文字列が配列であるかのように文字列をさし，そしてひとつの文字を取りだす．x の値が0であるならば，ASCII文字の '0' がプリントされる．x の値が10（0xa）であれば，その時ASCII文字の 'a' がプリントされる，等々．いいかえれば，x は配列のベース・アドレスからのオフセットを計算するのに使用され，そのオフセットにある文字が putchar( ) に送られる．配列のベース・アドレスは，2重の引用符表記の中に暗示されている．このやや奇妙にみえる式は，16進数からASCIIへの変換を行う．この式の簡潔さは魅力

注 22：次のようなものでリテラル文字列を変更することが可能である．

```
msg[2] = 'C';
```

しかし，リテラル文字列定数の内容を変更することは悪いプログラミング・スタイルだと考えられる．他の点では，同じ内容の2つのリテラル文字列が，メモリの別の場所に格納されている．例えば

```
char  *msg1 = "Hello";
char  *msg2 = "Hello";
```

は2つのポインタの大きさのオブジェクト（msg1, msg2）をつくる．同時に，文字列 "Hello" に初期化されている6バイトの文字配列も2つつくる．しかし，すべてのリテラル文字列がユニークであることを保証しないコンパイラが2, 3あることに注意すること．

注 23：次の文で4バイトの文字配列を宣言することもできる．

```
char  a[ ] = "foo";
```

しかし，これは思い違いをしやすい．この方法の利点のひとつは（char *p = "foo" と比較して），a が定数であり，シンボル・テーブルに格納されるということである．つまり，配列の最初の要素のアドレスを保持するために作られるポインタの大きさのオブジェクトがない．

的だが，プログラムは理解しにくいため，あまり維持しやすくはない．読みやすくするためには次のようにするのがよい．

```
char    a[ ] = { '0', '1', '2', '3', '4', '5', '6', '7',
                 '8', '9', 'a', 'b', 'c', 'd', 'e', 'f' };

putchar(  a[ x & 0xf ]  );
```

変数のはっきりした初期化の効果は，記憶クラスによって異なる．auto 変数（サブルーチンにローカルで，static と明示して宣言されていない変数）はサブルーチンに入るごとに初期化される．さらに，集合している型（配列や構造体）ははっきりと初期化されていないかもしれない．固定アドレスにある変数（すべてのグローバル変数と予約語 static で宣言されたローカル変数）は異なる扱われ方をする．固定アドレスにある集合体は，たとえそれらがサブルーチンにローカルであっても，初期化することができる．ローカルの static 変数は，連続したサブルーチン呼出しの間もその値を保っている．最初にサブルーチンが呼ばれるとき，静的なローカル変数は明示して初期化された値を持っている．同じサブルーチンが2度目に呼びだされたとき，その変数は，サブルーチンが最初にリターンしたときの値を保っている．

固定アドレスにある変数を初期化するための命令コードは生成されない．初期値はディスクから読み込まれる．つまり，プログラムの命令コードと，初期化されたデータは共にディスクに格納されている．固定したアドレスの場所を確保するために，値をディスク上におかなければならないから，明示して初期化されていないようなすべての変数はコンパイラが0に（ごみよりも）初期化する．これはコンパイル時間に2, 3ミリセコンド（1 000 分の1秒）余計にかかる．しかし，これは特に大きな配列では実際都合がよい．命令コードは auto 変数を初期化するときには生成される．

6.6 練　　習

次の線習問題では，配列の宣言以外には表記 p[i] は使ってはいけない．表記 *(p + i) はどこにも使用してはいけない（指標ではなく，ポインタを増加させるようにする）．

6-1　練習 5-2 の回答に，明示して配列の指標を使っているならば，ポインタを使ったプログラムに書き直しなさい．

6-2　文字列を逆順にするサブルーチンを書きなさい（ひとつの文字列を他にコピーしないこと）.

6-3　サブルーチン

```
subst( str, pattern, replacement )
char    *str, *pattern, *replacement;
```

を書きなさい．このサブルーチンはstrを通して，patternを探し，replacementでpatternをすべて置き換えなさい．例えば，呼出し

```
char    array[128] = "this foo is a foofooo string";
subst( array, "foo", "<->" );
```

は

```
"this <-> is a <-><->o string"
```

に変更された配列をもってリターンする．

7 高度なポインタ

7.1 複雑な宣言

C言語の利点のひとつは，その規則正しさである．同じ構文がポインタの宣言にも，使用にも用いられ，同じ優先順位と結合法則のルールが，どちらの場合にも適用される．演算子＋＋と＊の組合わせがどんなに重要であるかをみてきた．そして，同じことが複雑なポインタの宣言にも当てはまる．もっとも簡単な宣言をいままでは述べてきた．

```
int        *ip;      /* int へのポインタ */
int        ia[3];    /* int 配列         */
```

後者は，名前iaが配列の最初の要素をさし示すように初期化したポインタのごとく使用することができる（それが定数であることは除く．定数だから修正はできない）．上の2つの表記を組み合わせると何になるか？

```
int        *api[3];
```

が宣言しているものは何か？　配列か，それともポインタの配列か？　この質問に答える鍵は優先順位図（Appendix A）にある．どちらの演算子の優先順位が高いのか，＊か［ ］か？　優先順位図によると，［ ］は＊の上の行にあることがわかる．だから，［ ］のほうが高い優先順位を持つ．つまり，＊よりも名前apiとの結びつきが強い．この事実から，宣言にかっこ付けをすることができる．

```
int        (  *( api[3] ) );
```

apiが何であるかみつけるためには，もっとも内側のかっこ内から始めて，外側にすすむ．もっとも内側の式は，それだけが行にあれば，配列の宣言をしている

ことになる．したがってapiは配列であり，3つの要素を持つ．何の配列か？　ひとつ外側のかっこのレベルにすすむと，＊がある．＊だけが行にあったら，ポインタを宣言しているのだろう．それで，apiはポインタの配列である．では，何へのポインタか？　もっと外側にすすんで，予約語**int**をみつける．これをまとめると

```
int      *api[3];
```

は，int	int
へのポインタ	＊
を3個の要素として持つ，配列	[3]
である．	

intへのポインタの配列は宣言したが，int自体は宣言していない．適当に初期化された配列が図7･1に示されている（注1）．図7･1には（本章のほとんどの図には），すべてのセルのアドレスが数にコロンをつけて示してある．これらのアドレスは簡略形である．しかし，図で表しているものを示すには役にたつ．

図7･1では，apiの3つの要素が整数変数x，y，zをさし示すように初期化されている．これらの変数は，明示して宣言しなければいけない．xとyは，apiが宣言したときと同じ優先順位を使ってアクセスされる．[ ]は＊より高い優先順位である（＊より結合が強い）から，次の文でxを初期化することができる．

```
*api[0] = 1;        /* 1をxに入れる */
```

優先順位をあきらかにするために式にかっこをつけてもよい．

```
*( api[1] ) = 2;  /* 2をyに入れる */
```

演算子＊が式api[1]全体に作用して，api[1]にさし示されるオブジェクトが更新される．

驚くべきことは（少なくとも筆者にとっては），xは＊＊apiを使って参照することもできる．この場合，配列名は最初の要素へのポインタに評価されるので，＊apiは最初の要素の内容であり，＊＊apiは最初の要素がさし示すオブジェクトとなる．別の方法でこれを表すと，＊api ＝＝ api[0]で＊(api[0]) ＝＝ xであるから，＊＊api ＝＝ xとなる．

注1：配列がグローバルか，またはstaticで宣言されるのならば，宣言するときに初期化することができる．

```
int     x, y, z;
int     *api[3] = {&x, &y, &z};
```

```
int      *api[3];  /* int ポインタの配列と      */
int      x, y, z;  /* 配列の要素がさし示す      */
                   /* オブジェクトの宣言.       */

api[0] = &x;     /* 配列の初期化       */
api[1] = &y;
api[2] = &z;

*api[0] = 1;        /* 1をxに入れる.   */
*( api[1] ) = 2;    /* 2をyに入れる.   */
```

次のようにみえるだろう：

```
                              x:
                          100:|-------|
          api:            +-->|   1   |
          |-------|       |   |-------|
    400:  | 100 *-|---+          y:
          |-------|           200:|-------|
    402:  | 200 *-|----------->|   2   |
          |-------|                |-------|
    404:  | 300 *-|---+          z:
          |-------|   | 300: |-------|
                      +----->|   ?   |
                             |-------|

    api ==              配列の最初の要素のアドレス          == 400
   *api ==   api[0] ==  アドレス400にあるセルの内容         == 100
  **api == *api[0] ==  アドレス100にあるセルの内容         == 1
  **api == *api[0] ==  api[0][0]

          api[1] ==  apiからオフセット1にあるセルの内容
                     (メモリ番地402にあるセル)           == 200

         *api[1] ==  アドレス200にあるセルの内容         == 2
```

図 7・1　初期化されたintへのポインタの配列

apiの宣言を解釈した手順を一般化することができる.

（1）優先順位図を使って，宣言を十分にかっこ付けをする．オブジェクトの名前はもっとも内側にあるべきである.

（2）節を書いて名前から始める.（．．．は（3）の部分）

名前は，．．．である.

（3）もっとも内側のかっこから外側へ処理する.

みたもの：	次の説で置き換える：
*	への（複数の）ポインタ
[n]	のn個の要素を持つ，配列
( )	（もどる）

型の予約語に達するまで，本方法を続ける（最後に（3）の部分は逆に読んでまとめる）. 次に例をあげる.

```
int     aai[3][2]
```

は "2次元配列" として使うこともできるが, "2次元配列" ではない. 本当は何であるか理解するために, たった今定めたルールを適用する. 2組の '[ ]' はあきらかに同じ優先順位である. しかし, それらは左から右に結合が強い. 宣言は次のようにかっこで囲むことができる.

```
int     ( aai[3] )[2]
```

ルールに当てはめて, 内側から外にすすむ. 次の結果を得る.

aai は,
　を3個の要素として持つ配列, ┐
　を2個の要素として持つ配列, ├ この部分は下から上に読む.
　int ┘
である.

つまり, aai は複数の配列の配列である. メモリ中の配置は図7・2に示されている.

図7・2をみると2, 3気がつくことがある. 最初は, 代入文である. ip = (int *) aai; 配列名は, 最初の要素へのポインタに評価される. aai は配列の配列であるから, aai の最初の要素は, 2つの要素を持つ配列である. 一方, ip は, 2つの要素を持つ配列へのポインタとしてではなく, ひとつの **int** へのポインタとして宣言されている. この問題はキャストで回避する. キャストは, コンパイラに変数を使用する前に型変換をするように知らせる. aai (**int** の2つの要素を持つ配列へのポインタ) を単純な **int** へのポインタに変換したい. 変数名とセミコロンを取り除いた, 必要な型の変数宣言をかっこで囲んでキャストを形成する. 整数ポインタは

```
int     *ip;
```

のように宣言されるから, さきの手順でキャストをつくると,

```
(int    *)
```

になる. このキャストで aai を処理して, 一時的に aai の型を **int** へのポインタにかえる. aai の内容はかわらない. ただコンパイラの aai の取扱い方がかわる.

配列の最初をさし示すように初期化された ip は, 面白い方法で使うことができる. 整数がポインタを通して直接アクセスされるときに, コンパイラは実際にさし示されているオブジェクトが何かわからない. コンパイラが処理しなければならないのは, ひとつの要素へのポインタだけならば, 2×3の整数配列と6要素の整数配列を区別できない. aai が直接アクセスされるとき, コンパイラは何であ

```
#define ROWSIZE 3
#define COLSIZE 2

int     aai[ ROWSIZE ][ COLSIZE ];
int     *ip ;

ip = (int *) aai;
*(ip + 2) = 6;

        ip:
        |-------|
100:    | 200 * |
        |- - -|-|
              |
              |
        aai:  V
        |=======|
200:    |       |  aai[0][0] == *ip
        |- - - -|
202:    |       |  aai[0][1] == *(ip + 1)
        |=======|
204:    |   6   |  aai[1][0] == *(ip + 2) == *(ip + (COLSIZE * 1) + 0)
        |- - - -|
206:    |       |  aai[1][1] == *(ip + 3) == *(ip + (COLSIZE * 1) + 1)
        |=======|
208:    |       |  aai[2][0] == *(ip + 4) == *(ip + (COLSIZE * 2) + 0)
        |- - - -|
210:    |       |  aai[2][1] == *(ip + 5) == *(ip + (COLSIZE * 2) + 0)
        |=======|
        |       |
```

ip == 番地100にあるセルの内容 == 200

ip + 2 == ipはintをさし示すから，ip+2は4 (2*sizeof(int)) をipの内容に加える． 200+4 == 204

*(ip + 2) == 番地204にあるセルの内容 == 6

図 7・2 配列の配列

るのかわかる(この情報は宣言で規定されていたから). 式aai[1][1]と*(ip + 3) はどちらも同じセルを参照する．これは，配列をはやく初期化したいときには有効である．複合した配列のインデックス（[ ] を使用したもの）は，3つのかけ算が必要である（注2). そして，ポインタを増加するためにはかけ算は必要ない．だから，かなり効率がよい．aaiは，図7・3にあるループを使って初期化できる．

aaiのひとつの要素が，実際どのようにアクセスされるかみてみよう．aaiは2

注 2：aai[i][j] のアドレスをみつけるために，コンパイラは次の計算をしなければならない．

addr = aai + (i * COLSIZE * sizeof(int)) + (j * sizeof(int))

ここで，aaiは配列のベース・アドレスである．3回のかけ算と2回のたし算が必要である．

```
#define ROWSIZE 2
#define COLSIZE 3

int             aai[ ROWSIZE ][ COLSIZE ];
register int    *ip , i ;

for( i = ROWSIZE * COLSIZE, ip = (int *)aai;  --i >= 0;  *ip++ = 0)
        ;
```

図 7・3　簡単なポインタを用いた多次元配列の初期化

つの要素を持つ配列の配列であることを思い出しなさい．つまり，aai[x] はこの2つの要素を持つ配列のうちひとつをとる．ひとつの配列全体をとる．ただひとつの要素を取り出すためには，もうひと組の '[ ]' が必要である．

```
( aai[x] )[y]
```

は aai にあるひとつの整数（aai[x] の配列の y 番目の要素）を選びだす．'[ ]' は左から右に関係するから，かっこなしで aai[x][y] と書くことができる（注3）．しばらくの間，配列の配列の話題にもどる．他の例をみてみよう．

```
int     **ppi;
```

また，これにかっこ付けをすると，宣言は下のようになる．

```
int     *( *ppi );
```

内側から外側にみていって，次を得る．

ppi は，
　へのポインタ ┐
　へのポインタ ├─ 逆に読む．
　int ┘
である．

2つのアスタリスクに混乱されないように．特に，ppi を“**int** のポインタのポインタ”と思ってはいけない．これはことばの上では意味がないし，C 言語ではなおさら意味がない．ppi はひとつの，ポインタの大きさの，オブジェクトである．メモリ割当ての点から見ると，ppi が他のポインタをさし示していてもたいしたことではない．ppi と ppi を初期化するために必要なデータ構造を図 7・4 に示す．

注 3：前にも述べたように，かっこを用いた，配列へのランダム・アクセスはとても非効率的な操作である．さらに，大きな配列のアプリケーションの多くは，実際にはランダム・アクセスにする必要はない．読者は通常は順番に，行ごとか，または，列ごとに配列をアクセスするだろう．だから，いつでも2重のかっこよりも効率のよいポインタと定数を使用したアクセス方法を始めることができる．

```
int      i;
int      *pi;
int      **ppi;

ppi   = &pi;  /* ppi は int へのポインタのアドレスを得る  */
pi    = &i;   /* pi は int のアドレスを得る              */
**ppi = 2;    /* 2 を i に入れる                         */

         ppi:                    pi:                    i:
100:  |-------|     200:  |-------|    300:  |-------|
      | 200 *-|-------->| 300 *-|------->|   2   |
      |-------|           |-------|          |-------|


 &pi  ==  pi のアドレス  ==  200
  &i  ==  i のアドレス   ==  300

  ppi  ==  ppi の内容                  ==  200
 *ppi  ==  番地 200 にあるセルの内容   ==  300
**ppi  ==  番地 300 にあるセルの内容   ==  2
```

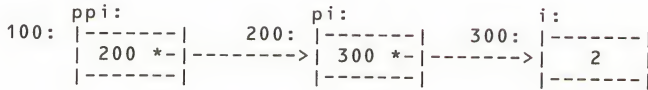
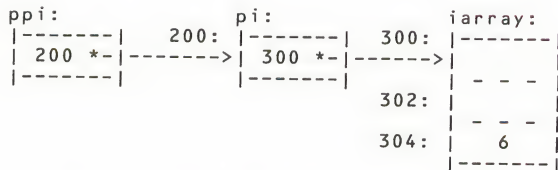



図 7・4 整数ポインタへのポインタ

いままでみてきたように，コンパイラはただのオブジェクトへのポインタと，オブジェクトの配列の最初の要素へのポインタとを区別できない．いいかえると，ポインタを操作するときには，さし示されているオブジェクトが何にみえるか区

```
int   iarray[3], **ppi, *pi ;

ppi = &pi;
pi  = iarray;
*( *ppi + 2 ) = 6;

ppi:                   pi:                  iarray:
|-------|    200:  |-------|   300:  |-------|
| 200 *-|------->| 300 *-|------>|       |
|-------|          |-------|         | - - - |
                               302:  |       |
                                     | - - - |
                               304:  |   6   |
                                     |-------|

      &pi          ==  pi のアドレス                      ==  200
   iarray          ==  iaray の最初の要素のアドレス       ==  300

      ppi          ==  ppi の内容                         ==  200
     *ppi          ==  アドレス 200 にあるセルの内容      ==  300
     *ppi + 2      ==  300+(2*sizeof(int))                ==  304
 *( *ppi + 2 )     ==  アドレス 304 にあるセルの内容      ==  6
```

図 7・5 ppi を使う 2 番目の方法

別できない．例えば，ppi は配列へのポインタへのポインタにもなることができる．この状態を図7·5に示す．

他の可能性もある．ひとつのオブジェクトへのポインタは，複数のオブジェクトの配列をさし示すこともできる．前の例の中間のポインタ（pi）もまた配列である（図7·6参照）．規則はないのだろうか？　コンパイラは，何でも追跡するわけではない．あるポインタが，配列をさしているのかいないのかおぼえておくのはプログラマの責任である．

```
int   array_of_ints[3]
int   *array_of_ptr[3]
int   int0, int1;
int   **ppi;

ppi = array_of_ptr;
array_of_ptr[0] = &int0;
array_of_ptr[1] = &int1;
array_of_ptr[2] = array_of_ints;

**(ppi + 1) = 6;
*( *(ppi + 2) + 2) = 7;

                                                int0:
                                         200: |-------|
ppi:               array_of_ptr:         +---->|       |
|-------|          100: |-------|        |     |-------|
| 100 *-|-------->|  200 *-|---+          int1:
|-------|               |-------|       300: |-------|
                   102: | 300 *-|----------->|   6   |
                        |-------|            |-------|
                   104: | 400 *-|---+
                        |-------|   |  array_of_ints:
                                    +-->|-------|
                                  400:  |       |
                                        | - - - |
                                  402:  |       |
                                        | - - - |
                                  404:  |   7   |
                                        |-------|

          ppi        ==  ppiの内容                    == 100
          ppi + 1    ==  100+sizeof(int)              == 102
       *(ppi + 1)    ==  アドレス102にあるセルの内容    == 300
      **(ppi + 1)    ==  アドレス300にあるセルの内容    == 6

      ppi                ==  ppiの内容                 == 100
      ppi + 2            ==  100+(2*sizeof(int))       == 104
    *(ppi + 2)           ==  アドレス104にあるセルの内容 == 400
    *(ppi + 2) + 2       ==  400+(2*sizeof(int))       == 404
*( *(ppi + 2) + 2)       ==  アドレス404にあるセルの内容 == 7
```

図 7·6　ppi を使用するもうひとつの方法

ポインタへのポインタの別の使い方を考えてみよう．図7·7には，main( )(注4)の暗黙の引数argvとargcの配置が示してある．配列vectorsは通常，プログラムがブートされるときにrootモジュールにつくられる．vectorsは，コマンドの引数から組み立てられて，argvでmainに渡される．vectorsの宣言は次のようにかっこ付けされる．

```
char     *(vectors[ ])...
```

内側から外にみていって

```
vectors は,
    (不定長の) 配列 ┐
    へのポインタの  ├─ 逆に読む.
    char           ┘
である.
```

配列の長さは初期化で定義される．初期化の並びに3つのオブジェクトがあるから，vectorsは3要素長である．これらのオブジェクトのおのおのは各文字列の最初の文字へのポインタに評価される．変数vectorsは3つの文字配列ではなく．3つのポインタに初期化される．vectorsはポインタの配列であるから，その名前はポインタへのポインタに評価される．それで，argvへの代入は難しくなる(argvもまたポインタへのポインタであるから)．最後に，argcはargvにある要素の数を保持している．式

```
sizeof(vectors) / sizeof(*vectors)
```

は，バイトで表した配列の大きさを，バイトで表した配列の最初の要素の大きさで割ると，要素の数になる．

argvのいくつかの使用例を，図7·7の下のほうにあげている．argvはvectorsの最初の要素へのポインタであり，配列名は配列の最初の要素へのポインタであるから，vectorsとargvはこれらの例では互いに交換して使用できる．

＊argvはargvにさし示されるオブジェクトで，vectors[0]の内容である．

注 4：main( )の引数を使っても使わなくても，main( )は必ず2つの引数をスタックにおいている．後でそれらをどのように使用するのか調べる．最近のコンパイラには第3の引数envpがある．この最後の引数は，文字列ポインタの配列へのポインタである．各文字列ポインタは環境変数の文字列を保持している．文字列ポインタの配列の最後の要素はNULLである(つまりそこには環境変数はない)．引数は次のように宣言されている．

```
main( argc, argv, envp )
int     argc;
char    **argv;
char    **envp;
```

```
char    *vectors[ ] = { "first", "second", "third" };
char    **argv = vectors;
int     argc   = sizeof(vectors)/sizeof(*vectors) ;

                                        |-------------------------|
argv:               vectors:      +-->|'f'|'i'|'r'|'s'|'t'|'\0'|
|-------|           |-------|     |    |-------------------------|
|   *---|------->|   *---|---+
|-------|           |-------|          |------------------------------|
                    |   *---|------->|'s'|'e'|'c'|'o'|'n'|'d'|'\0'|
argc:               |-------|          |------------------------------|
|-------|           |   *---|---+
|   3   |           |-------|     |    |-------------------------|
|-------|                         +-->|'t'|'h'|'i'|'r'|'d'|'\0'|
                                       |-------------------------|
```

argv は vectors の最初の要素をさし示しているから，次の 5 つの例のように
argv と vectors は互いに交換できる．

```
    *argv           ==  vectors[0]                     ==文字列 "first" へのポインタ
   **argv           ==*vectors[0]      ==argv[0][0] ==文字 'f'
   * (argv+1)       ==  vectors[1]                     ==文字列 "second" へのポインタ
  ** (argv+1)       ==*vectors[1]      ==argv[1][0] ==文字 's'
 * (* (argv+1)+2)==*(vectors+1)[2]==argv[1][2] ==文字 'c'
```

図 7・7　argv

argv と vectors は交換可能であるから，argv[0] とすることができる．[ ] 表記はポインタにオフセットを加えて計算されたアドレスにあるオブジェクトをとり出すように，コンパイラに指示しているだけである．argv は文字列ポインタであるから，puts(*argv) を呼出して文字列 first をプリントすることができる．

次の式は argv にオフセット 1 を加えて vectors の 2 番目の要素をアクセスする．*(argv + 1)，*(vectors +1), argv[1]，vectors[1] も同様である．式 puts（*(argv +1)) は文字列 second をプリントする．argv は変更されない．たし算は一時的な変数で行なわれる．

argv は vectors の最初の要素の内容に評価されるから，（*argv)（かっこは必要ない）は vector の最初の要素によってさし示されているオブジェクト，"first" の文字 'f' になる． 同様に，*(argv +1) は配列 vectors の 2 番目の要素であるから，**(argv + 1) は vectors の 2 番目の要素，"second" の文字 's' になる．

最後は*(*(argv + 1) + 2) である．内側から外へ処理すると，*(argv + 1) は，オブジェクトひとつ分のオフセットが argv に加えられた後，argv にさし示されるオブジェクトである．それは vectors の 2 番目の要素（vectors[1]）で

図 7・8 *(*(argv+1)+2)

```
           argv:
           |--------|
           | 100 *-|---+
           |--------|   |
                        |
                |---V---|
     100:       |       |
                |-------|          |-----|
     102:       | 200 *-|--------->| 's' | 200
                |-------|          |-----|
     104:       |       |          | 'e' | 201
                |-------|          |-----|
                                   | 'c' | 202
                                   |-----|
                                   | 'o' | 203
                                   |-----|
```

argv	==	argv の内容
*argv	==	番地 100 にあるセルの内容
argv+1	==	argv の内容 + (1*sizeof(char *))
	==	100 + 2
	==	102
*(argv+1)	==	番地 102 のセルの内容
	==	200.
*(argv+1) +2	==	200 + 2
	==	202
*(*(argv+1) +2)	==	番地 202 のセルの内容
	==	'c'

図 7・9 *(*(argv+1)+2) Shown With Addresses

ある．vectors[1] は，"second" の最初の文字（'s'）へのポインタである．このポインタに2を加えて "second" の3番目の文字（'c'）をさすポインタをつくる．しかし，いままでのところでは 'c' へのポインタを得ただけである．もっとも左に＊を加えると 'c' 自体になる．それで，putchar(＊(＊argv ＋ 1) ＋ 2)) は 'c' をプリントする．この例は図7·8に詳しく書いている． 図7·9は同じ例だが，実際のアドレスを入れてある．

＊(ptr ＋ i) は ptr[i] で表されるから，この文はかなり簡単にできる．そのルールを1回適用すると，次のようになる．

```
*(      ptr     +i) ==      ptr     [i];
*( *(argv+1)   +2) ==  *(argv+1)  [2]
```

もう1回それを適用すると，次のようになる．

```
*(      ptr     +i)         ==      ptr   [i];
*(     argv     +1) [2]     ==    argv    [1]   [2]
```

面白いことに，2つのポインタを通して文字列にアクセスするために，2次元配列をアクセスするためによく使ったものと同じ表記のしかたを使用している．しかし，argv は2次元配列とはまったく違う．だからなおさら配列自体は本当に存在しないということになる．

argv は変数だから，vectors のさまざまな内容と同様に，変更することができる．図7·10に式

```
++ * ++argv;
```

の結果を示す．この式は，かっこで囲むと＋＋(＊(＋＋argv)) になり，2つのことを行う．もっとも内側の＋＋は argv に接しているから，argv は増加して，vectors[1] をさし示す．前置きインクリメント（識別子のまえに＋＋がある）だから，続くすべての動作は増加後の argv の値を使う．＊が次に評価され，外側の＋＋は＊argv（増加後の）に適用される．再び前置きインクリメントであるが，それにより増加した argv にさし示されているオブジェクト自体が増加する（今度は "second" の 'e' がさし示される）．'e' をそのポインタでアクセスするには，もうひとつ＊が必要である．この動作を図7·10に示す．

図7·10の下方にある例の中で，まだみたことがないものがひとつだけある．その式＊＊(argv － 1) は，argv の現在の値から1オフセットひいて，vectors[0] をさし示すポインタをつくる．すなわち，＊(argv － 1) は (argv － 1) に

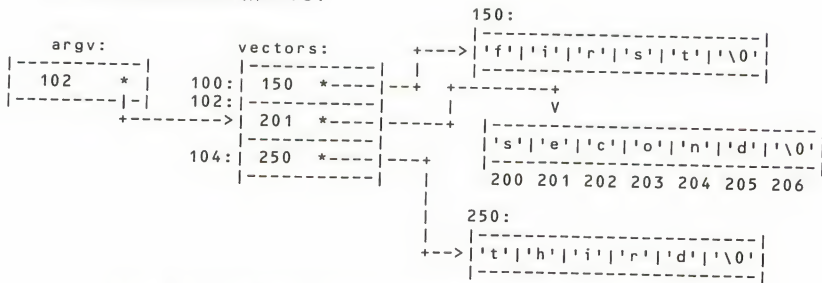
さし示されるオブジェクトであり，そのオブジェクトはargvから1オブジェクトの大きさをひいたアドレスにある．式**(argv − 1）はvectors[0] にさし示されるオブジェクト，"first" の 'f' である．−1は '[ ]' の中にも使用できて（*argv[−1]，argv[−1][0] にあるように），負のオフセットを表す．

図7·10の最後の例は，−1のさきの例とは違った使い方を示している．これは−1をargv自体にではなく，*argv（argvがさしているオブジェクト）に適用している．*argvは "second" の 'e' へのポインタであるから，（*argv）− 1は 'e' の前の文字 's' をさすポインタである．つまり*((*argv) − 1）は，そのアドレスにあるオブジェクト 's' である．

次の命令を実行する

```
        ++ * ++argv      ==      ++( *( ++argv ) )
```

argv と vectors は次の関係がある．

```
                                                150:
                                                |-------------------------|
  argv:                vectors:            +--->|'f'|'i'|'r'|'s'|'t'|'\0'|
|------------|         |------------|  |        |-------------------------|
|  102     * |   100:  | 150   *----|--+   +---------+
|----------|-|   102:  |------------|      |         V
           +---------->| 201   *----|------+   |-------------------------------|
                       |------------|          |'s'|'e'|'c'|'o'|'n'|'d'|'\0'|
                 104:  | 250   *----|---+      |-------------------------------|
                       |------------|   |       200 201 202 203 204 205 206
                                        |
                                        |     250:
                                        |     |-------------------------|
                                        +-->|'t'|'h'|'i'|'r'|'d'|'\0'|
                                              |-------------------------|
```

argv	100（vectors[0] のアドレス）を含んでいるところから始まる
++argv	100から102にargvを増加する（ポインタの計算が行われた），アドレス102にあるセル（vectors[1]）をさすようになる．
* ++argv	増加した後さし示されているオブジェクトの内容，番地102にあるセルの内容．
++ * ++argv	このセルを200から201に増加する．'e' をさすようになる．
* ++ * ++argv	アドレス201にあるセルの内容，'e'

この時点（全部の増加が終わった後）で，次の恒等式が成り立つ．

```
  *argv          ==   vectors[1]      == 文字列 "second" へのポインタ
 *(argv+1)       ==   vectors[2]      == 文字列 "third" へのポインタ
 **argv          ==  *vectors[1]      == argv[0][0]  ==  文字 'e'
 **(argv+1)      ==  *vectors[2]      == argv[1][0]  ==  文字 't'
*(*(argv+1) +2)  == *(vectors+2)[2]   == argv[1][2]  ==  文字 'i'
**( argv - 1)    ==  *vectors[0]      == argv[-1][0] ==  文字 'f'
*((*argv) -1)    ==  vectors[1][-1]   == argv[0][-1] ==  文字 's'
```

図 7·10　argvのインクリメント

別の例をみてみよう．

```
int     (* pai)[3] ;
```

ここには，宣言の一部としてかっこが使われている．まず，もっとも内側のかっこをみてみると，次のことがわかる．

pai は，
　へのポインタ，
　の 3 つの要素を持つ配列　— 逆に読む．
　int
である．

また，ポインタは宣言されているだけである（結局は配列をさし示すのだが，まだ今のところはそうではない）．pai が図 7・11（注 5）で初期化されている．

```
int     (* pai)[3];        /* ポインタの宣言  */
int     ai[3];             /* 配列の宣言      */

pai          = ai;         /* 初期化          */
**pai        = 0;          /* 使用            */
(*pai)[1]    = 1;
*(*pai + 2)  = 2;
```

次のようにみえる．

```
                 +
    pai:         |        ai:
|-------|        |  100: |-------|
| 100 *-|-->     |       |   0   |    ai[0] == (*pai)[0] == **pai;
|-------|        |       |- - - -|
                 |  102: |   1   |    ai[1] == (*pai)[1] == *( *pai + 1 )
                 |       |- - - -|
                 |  104: |   2   |    ai[2] == (*pai)[2] == *( *pai + 2 )
                 +       |-------|
```

図 7・11　int の配列へのポインタの初期化

pai は配列へのポインタであるから，そのさし示されるオブジェクト（∗pai）は，配列の 1 要素ではなく，配列全体である（注 6）．その配列の 1 要素にアクセ

注 5：次のように ai と pai を初期化することもできる．

```
int     ai[3] = {0, 1, 2};
int     (* pai)[3] = ai;
```

注 6：このため，多くのコンパイラ（例えば，Microsoft compiler）は次のような代入文には文句をいう．

```
pai = ai;
```

ここで問題なのは，配列名である ai が配列の最初の要素へのポインタに評価されること，つまり，int へのポインタに評価されていることである．pai は int 1 個へのポインタではなく，配列へのポインタであるから，コンパイラは型が違うと思う．この問題はキャストを用いて回避することができる．いいかえれば，ai を 3 つの要素も配列へのポインタにキャストする．

```
pai = (int (*)[3])ai;
```

スするには，pai ではなく，＊pai にインデックスをつけなければならない（[ ] 表記を加える）.[ ] はアスタリスクより高い優先度を持つから，かっこが必要である．ai[1] は（＊pai)[1] でアクセスできる．

これを一段すすめてみよう．配列の最初の要素（ai[0]）は，（＊pai)[0] でアクセスすることができる．ポインタの右に [0] があるが，その表示を，ポインタの左に＊をおく表示にかえることができる．したがって，（＊pai)[0] は＊＊pai でもアクセスできる．たった今適用したルールは

```
p[i] == *(p + i)
```

の変形した場合だから，この等式の p を（＊pai）で置き換えて，次式を得ることができる．

```
(*pai)[i] == *( (*pai) + i )
```

再び異なる角度からこの例をみてみよう．C 言語ではすべての式は何かに評価され，その何かは限定された型である．pai はその内容 100 に評価される．pai は **int** の配列へのポインタである．＊pai は int の配列へのポインタによってさし示されているオブジェクトである．＊pai は "int の配列" という型である．配列は通常名前で参照され，その名前は配列の 1 要素へのポインタに評価される．この例では＊pai は配列全体を参照している．だから，＊pai は配列名のように，配列の最初の要素へのポインタ，その最初の要素のアドレス 100，に評価される．pai と＊pai は同じ数，100 に評価される．しかし，pai 自体は **int** の配列へのポインタ型であり，一方＊pai は int の配列型である．＊pai は int の配列型であるから，＊＊pai は **int** 型であり，それは配列の最初の要素に評価される．

pai のもうひとつの特徴がある．さし示されているオブジェクトは配列全体であるから，そのオブジェクトの大きさは配列全体のバイトでの大きさである．ポインタ計算は pai がインクリメントされるときに行われ，さし示されているオブジェクトの大きさ（配列全体の）が pai の内容に加えられる．2 バイトの **int** だとすると，3 つの要素を持つ整数の配列の大きさは 6 である．pai をインクリメントすると，図 7·12 のように配列全体をとび越えてしまうことになる．この動作は，配列の配列に適用されるとき意味を持つようになる（図 7·13 をみよ)．pai はそこでインクリメントされ，最初の副配列を通り越して 2 番目の副配列をさしている．

pai について今おぼえたことを 2 次元配列に応用してみよう．aai[x]（ひと組

```
int     (* pai)[3];      /* ポインタの宣言  */
int     ai[3];           /* 配列の宣言      */
++pai;                   /* 次のことを行う  */

        pai:             ai:
        |-------|        |-------|
        |   *---|--+     |   0   |
        |-------|  |     |- - - -|
                   |     |   1   |
                   |     |- - - -|
                   |     |   2   |
                   |     |-------|
                   +--->|        |
```

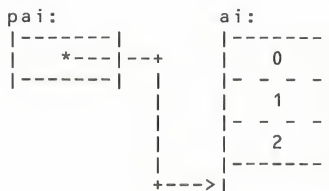



図 7・12　int の配列へのポインタをインクリメントする

```
int     (* pai)[3];      /* ポインタの宣言  */
int     aai[2][3];       /* 配列の宣言      */
++pai;                   /* 次のことを行う  */

pai:               aai:
|-------|          |-------|
|   *---|--+       |       |   aai[0][0]
|-------|  |       |- - - -|
           |       |       |   aai[0][1]
           |       |- - - -|
           |       |       |   aai[0][2]
           |       |=-=-=-=|
           +--->|  |       |   aai[1][0] == (*pai)[0] == **pai
                   |- - - -|
                   |       |   aai[1][1] == (*pai)[1]
                   |- - - -|
                   |       |   aai[1][2] == (*pai)[2]
                   |-------|
```

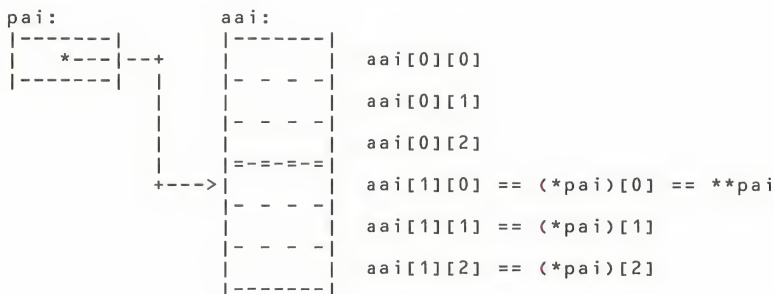



図 7・13　配列の配列に使用されている int の配列へのポインタ

の [ ] だけがある）は 3 要素の配列全体を参照する．だから，3 要素の配列へのポインタであるかのように使用できる．pai は，次の文で配列の 2 番目の要素をさすように初期化できる．

```
pai = aai[1] ;
```

いいかえれば，aai は複合配列である．ひと組の [ ] をもつ配列名は，副配列全体へのポインタに評価される．配列全体への参照は，その配列の最初の要素へのポインタに評価されるから，aai[i] も int へのポインタのポインタ型である．さらに，この部分的な参照は，それだけでポインタとして使用できる．いいかえれば，恒等式

```
*(p + i)  ==  p[i]
```

を

```
p[ ]...[ ][i] ==  (p[ ]...[ ]) [i]  ==  *( (p[ ]...[ ]) + i )
```

に拡張することができる．別の方法をみてみよう．恒等式は [ ] がなくなるまで続けて適用できる．

```
aai[i][j]  ==  *( (aai[i]) + j)  ==  *( *(aai + i) + j )
```

さまざまな関係が図7·14に示されている．3要素の配列への，どのポインタもこれらの例ではaaiに置き換えて使用できる．

すべてのかっことアスタリスクは，[ ] を用いた表記の実用性の核心を突いている．筆者が思うには，aai[1][2] のほうが*(*(aai + 1) + 2) よりも読んでわかりやすい．一方で，2番目の式の方がコンパイラの実際の動作に近い．

```
#define ROWSIZE 2
#define COLSIZE 3

int     aai[ ROWSIZE ][ COLSIZE ];

aai[i][j]  ==  *( (aai[i]) + j)  ==  *( *(aai + i) + j )

aai:
|-------|
|       |  aai[0][0] == *(  aai[0] )      ==  **aai ;
|- - - -|
|       |  aai[0][1] == *( (aai[0]) + 1 ) ==  *( *aai        + 1 )
|- - - -|
|       |  aai[0][2] == *( (aai[0]) + 2 ) ==  *( *aai        + 2 )
|=-=-=-=|
|       |  aai[1][0] == *(  aai[1] )      ==  *( *(aai + 1) + 0 )
|- - - -|
|       |  aai[1][1] == *( (aai[1]) + 1 ) ==  *( *(aai + 1) + 1 )
|- - - -|
|       |  aai[1][2] == *( (aai[1]) + 2 ) ==  *( *(aai + 1) + 2 )
|-------|
```

図 7·14 配列の配列

7.2 関数ポインタ

C言語は別の種類のポインタ（関数ポインタ）をサポートしている．ちょうど

```
int     (* array)[ ];
```

は**int**へのポインタを宣言しているように

```
int     (* pfi)( ) ;
```

は**int**をかえす関数へのポインタを宣言している．どちらの宣言にも，かっこが

必要である．かっこがなければ，array はポインタの配列になってしまうだろうし，pfi は，**int** へのポインタをかえす関数になってしまうだろう．みてきたように，配列名は，いつも配列の最初の要素のアドレスに評価される．同様に，関数名はいつも関数の最初の命令コードのアドレスに評価される．このアドレスは関数ポインタにおくことができ，そのポインタは，関数を直接呼びだすときに使うことができる．図7·15では，それがどのように行われるかを示している．

```
int      foo( a, b )
{
         /* ... */
}

main( )
{
         int      (* pfi)( );
         int      x, y;

         /* ... */

         pfi = foo;                   /* fooへのポインタpfiを初期化する */
         (* pfi)( x, y );             /* foo(x,y); と同じ              */
}
```

図 7·15　関数へのポインタ

代入文 pfi ＝ foo には，foo に続くかっこがないことに注意しなさい．かっこがあれば，foo が呼びだされて，pfi は foo(　) がかえすどんな値でも保持するように初期化されるだろう．かっこがなければ，サブルーチン名はそのサブルーチンのアドレス（[　] なしの配列名が，その配列のアドレスに評価されるように）に評価され，そのサブルーチンは呼びだされない（注7）．ある関数が次の文でポインタを通して直接呼びだされる．

```
(* pfi)( args );
```

"pfi にアドレスが含まれている関数を呼びだす"．ここでもかっこは必要である．次の例は，その理由をもっとも簡単に示している．

```
extern  int     *funct( );

x = *funct( );
```

ここでは，宣言は次のようにかっこ付けできる．

注 7：初期の Lattice C はバグのため，pfi ＝ &foo と書かざるを得ない．
　　実際は & は必要ない．

```
extern  int     *( funct( ) );
```

それで，functはintへの（ポインタをかえす）関数である．文x＝ ＊funct( )はfunct( ) がかえすポインタがさしているオブジェクトの内容をxに入れる．つまり，funct( ) はintのアドレスをかえし，＊funct( ) はそのintに評価される．

関数ポインタは，関数の配列をとることができない点で，他のポインタとは異なる（関数ポインタの配列はできるが，関数自体の配列はできない）．したがって，関数ポインタをインクリメントすることは規則違反になる（実際には，関数ポインタでのすべての算術演算が規則違反になる）．

関数名にかっこがつけられていないときには，いつも関数ポインタであることが暗に示されていることに注意する必要がある．例えば，サブルーチンlaurel( ) とhardy( ) は同じ引数を持ち，次のように書ける．

```
extern int      laurel( ), hardy( );
( *(condition ? laurel : hardy)  )( arg1, arg2, arg3 );
```

condition（条件）が真であるかどうかに関係して，laurel( ) かhardy( ) が呼びだされる．どちらでも，同じ引数が呼びだされた関数に渡される．かっこのない関数名が引数であるので，条件自体は関数へのポインタに評価される．＊が関数ポインタを通して，直接その関数を呼びださせる．さきの例は次のように書き直すことができる．

```
extern int      laurel( ), hardy( );
int             (* ptr)( );

ptr = condition ? laurel : hardy ;

( *ptr  )( arg1, arg2, arg3 );
```

ここではサブルーチン呼出しのかっこも演算子である．そのかっこはサブルーチン名の一部ではない．つまり，かっこ演算子はサブルーチンの呼出しを生成するために，どの関数へのポインタにもつけることができる．配列名が，その配列の最初の要素へのポインタに評価されるように，サブルーチン名は，関数へのポインタに評価される．[ ] が，配列要素へのどんなポインタにもつけることができるように，( ) も関数へのどんなポインタにもつけることができる．

関数ポインタは，使い方がいくつかある．その使い方は，ジャンプ・テーブルやルックアップ・テーブル（構造体の配列，その構造体のひとつのフィールドが

```
 1 void     argv_sort( argc, argv )
 2 int      argc;
 3 char     **argv;
 4 {
 5          /* シェル・ソートを使用してargvをソートする */
 6
 7          register int    i, j;
 8          int             gap, k;
 9          char            **p1, **p2, *tmp ;
10
11          for( gap = argc >>1 ;  gap > 0  ; gap >>=1 )
12                  for( i = gap; i < argc; i++ )
13                          for( j = i-gap; j >= 0 ; j -= gap )
14                          {
15                                  p1 = argv +   j;
16                                  p2 = argv + ( j+gap );
17
18                                  if( strcmp( *p1, *p2 ) <= 0 )
19                                          break;
20
21                                  tmp = *p1;      /* 2つの要素を交換する */
22                                  *p1 = *p2;
23                                  *p2 = tmp;
24                          }
25 }
```

図 7・16　汎用目的でないシェル・ソート

キーで，もうひとつのフィールドはそのキーに出会ったとき呼びだされる関数へのポインタになっている)，オブジェクト志向プログラミング（関数が特定のオブジェクトに対して特別な情報を持ち，その情報を関数へのポインタを通してサブルーチンに渡すことができる）にある．

この最後の使用法のよい例は，ssort( )，UNIXのqsort( )をモデルにした汎用のシェル・ソート・サブルーチンである．汎用目的のssort( )をみる前に，argvをソートするためにつくられたプログラム，argv_sortをみてみよう（図7・16参照)．

もしシェル・ソートがどのように動くのかよく知らなければ，このまま続ける前にAppendix Bにある記述をみるべきである．

argv_sortは，argvをソートするためにつくられたシェル・ソートである．p1とp2は，1回通す間に考えられる2つの要素へのポインタである．この2つのポインタは，図7・16の15, 16行で初期化されている．ポインタの計算が行われて，argv ＋ j はargv[j] のアドレスに計算される．同様に，argv ＋ (j ＋ gap) はargv[j ＋ gap] のアドレスに計算される．2つの要素は，strcmp( ) を使って比べられる．strcmp( ) は，＊p1 ＜ ＊p2ならば負の値をかえし，＊p1 ＝＝ ＊p2ならばゼロをかえき，＊p1 ＞ ＊p2ならば正の値をかえす．p1とp2はargv 配

列へのポインタである．したがって，それらは文字ポインタへのポインタである．strcmp( ) は引数として単純な文字ポインタが必要である．だから，間接的レベルはstrcmp( ) で1レベル取り除かれていなければならない．2つの要素は21～23行で交換される．

基本的なシェル・ソートを真の汎用目的にすることができる（つまりどんな配列も，ポインタの配列，構造体の配列等，ソートすることができる）．argv_sort( ) にこのような能力を与えるには2つの変更が必要である．第一に，配列のひとつのセルの大きさをバイト数で教える必要がある．argv_sort( ) には，この情報は配列宣言に暗示されている．つまり，文字配列へのポインタとして定義されているから，コンパイラは配列の各要素が文字ポインタの大きさであることを知って，ポインタ計算を正しく行うことができる．汎用目的のルーチンは，コンパイル時にひとつの要素の大きさを知ることはできず，実行時までその情報がないからポインタ計算をすることができない．だから，そのソート・ルーチンにはひとつの要素の大きさをパラメータとして渡し，ポインタ計算をしなければならない．

汎用目的のルーチンに必要な第2の情報は，比較ルーチンへのポインタである．argv_sort( ) は，argv をソートしていることがわかっていたから，直接strcmp( ) を呼びだすことができた．ssort( ) は，ソートされるオブジェクトの大きさ以外その情報を何も知らないから，strcmp( ) を直接呼びだすことはできない．strcmp( ) にかわる比較ルーチンが必要である．配列の要素がどのようなものかを知り，その2つの要素を比較することができるルーチンが必要である．この比較関数はソートされる配列の種類各々に対して代わり，その関数へのポインタ

```
Given:
        int     array[ 10 ];

        cmp( ip1, ip2 )
        int     *ip1, *ip2;
        {
                return *ip1 - *ip2 ;
        }
```

次の文で配列はソートできる：

```
        ssort( array, 10, sizeof(int), cmp );
```

図 7・17 int の配列をソートするための ssort の使用

をソート・ルーチンに渡す．ソートしたい配列の種類によって異なる比較関数を書かなければならない．その比較関数は，2つの配列要素へのポインタを渡されることができ，それ以外はstrcmp( )のように動くべきである．

ssort( )でargvをソートするとき，比較関数はargvの要素への2つのポインタを渡される．間接を1段取り除き，strcmp( )を呼びだして比較する．intの配列をソートするとき，比較関数は**int**へのポインタを渡され，int自体を比較する(ポインタを通して関接的に)．構造体の配列をソートするときは，比較関数は

```
cmp( cpp1, cpp2 )
char    **cpp1, **cpp2;
{
        return strcmp( *cpp1, *cpp2 );
}


main( argc, argv )
int     argc;
char    **argv;
{
        ssort( ++argv, --argc, sizeof(*argv), cmp );

        while( --argc >= 0 )
                printf("%s\n", *argv++ );
}
```

図 7・18　argvをソートするためのssort( )の使用

```
Given:
                typedef struct
                {
                        int     key, element1, element2, etc;
                }
                ELEMENT;


                ELEMENT array[10];


                compare( e1, e2 )
                ELEMENT *e1, *e2;
                {
                        return( e1->key - e2->key );
                }
```

次の文で配列はソートできる：

```
ssort( array, 10, sizeof(ELEMENT), compare );
```

図 7・19　構造体の配列をソートするためのssortの使用

構造体への2つのポインタを渡され，構造体のキー・フィールドを比較する．この3つの状態が図7·17，7·18，7·19に示されている．ssort(）は図7·20に示されている．

今述べた変更がここにはすべて組み込まれている．baseは2行目で文字ポインタとして宣言されている．**char**は1バイト幅であるから，文字ポインタのポインタ計算は普通の計算と同じである．**char**へのポインタにさし示されているオブジェクトの大きさは1である．それで，p1とp2を初期化するとき(29，30行目)，

```
 1 void     ssort( base, nel, width, cmp )
 2 char     *base;
 3 int      nel, width;
 4 int      (*cmp)( );
 5 {
 6          /*       汎用目的シェル・ソート：
 7           *
 8           *       Base .. は格納される配列のベース・アドレス
 9           *       nel ... 配列にある要素の数
10           *       width . ひとつの要素のバイトでの大きさ
11           *       cmp ... 比較する関数へのポインタ，
12           *               次の式で呼びだされる．
13           *                       x = (*cmp)( a, b )
14           *               ここで，aとbは2つの要素へのポインタ．
15           *               次のようにリターンする．
16           *                       x <  0   ならば，  a <  b
17           *                       x == 0   ならば，  a == b
18           *                       x >  0   ならば，  a >  b
19           */
20
21          register int    i, j;
22          int             gap, k, tmp ;
23          char            *p1, *p2;
24
25          for( gap = nel >>1 ;  gap > 0  ; gap >>=1 )
26                  for( i = gap; i < nel; i++ )
27                          for( j = i-gap; j >= 0 ; j -= gap )
28                          {
29                                  p1 = base + ( j       * width);
30                                  p2 = base + ((j+gap) * width);
31
32                                  if( (*cmp)( p1, p2 ) <= 0 )
33                                          break;
34
35                                  for( k = width; --k >= 0 ;)
36                                  {
37                                          tmp    = *p1;
38                                          *p1++ = *p2;
39                                          *p2++ = tmp;
40                                  }
41                          }
42 }
```

図 7·20　ssort( )：汎用目的のソート・ルーチン

配列の1要素の大きさとしてwidthをパラメータに使って，ポインタ計算をすることができる．argv_sort( )では，コンパイラが通常のポインタ演算を使って初期化していた．strcmp( )を使う代わりに，ssort( )はさきに説明した比較関数を使う，32行目で直接それを呼びだす．最後に，2つのオブジェクトは間接的に1バイト交換されなければならない（実行時まで1要素の大きさがわからないから）．この交換は35〜40行目にあるforループで行われる．

7.3 逆の手順

いままで複雑な宣言を解釈するしかただけをみてきた．このセクションは，その逆のやり方の例をあげ，変数の宣言をする．手順はいままで使っていたものと同じだがすべての段階が逆転する．変数を宣言するには

（1） 何を宣言しようとしているか書く．データ構造が複雑ならば，図を書く．これは，何をしようとしているかわかるのに役立つ．図には，実際に宣言するオブジェクトの名前を丸で囲む．

（2） オブジェクトの名前を書きだす．そのオブジェクトがポインタであれば，名前の横にアスタリスクを書く．配列ならば，右横に[ ]をかき，その中に数を入れる．最後に，書いたもの全体をかっこで囲む．

（3） この手順を続け，名前から始める．図を書いていたらポインタが参照する矢印の方向へ進む．

2, 3の例をあげる．**int**の配列へのポインタの配列へのポインタである変数名papaは，次のように宣言される．

`papa`	名前を書く．
`(*papa)`	ポインタである．それでアスタリスクとかっこを加える．
`((*papa)[ ])`	papaは何へのポインタか？　配列．それでは’[ ]’とかっこを加える．
`(*((*papa)[ ]))`	何の配列か？　ポインタ．それで，もうひとつアスタリスクとかっこを加える．
`(*((*papa)[ ]))[ ]`	何へのポインタか？　配列．それでは，2組目の’[ ]’が必要である．
`int (*((*papa)[ ]))[ ]`	何の配列か？　int，それでは単語intを書いて，終わり．

この例では，アスタリスクが'[ ]'より優先度が高いから，すべてのかっこが必要である．

別の例は，**long** へのポインタをかえす関数へのポインタの配列へのポインタである too_complex という変数を宣言するために，次のことを行う．

`too_complex`	名前を書く．
`(*too_complex)`	ポインタである．それで，アスタリスクを加える．
`((*too_complex)[])`	配列へのポインタである．それで，'[ ]'を加える．
`(*((*too_complex)[]))`	ポインタの配列である．それで，別のアスタリスクを加える．
`(*((*too_complex)[]))()`	関数へのポインタである．それで関数を示す一組のかっこを加える．
`*(*((*too_complex)[]))()`	ポインタをかえす関数である．それで，もうひとつアスタリスクを加える．関数に必要な（ ）は＊より優先順位は高いから，ここにもう一組のかっこは必要ない．
`long  *(*((*too_complex)[]))()`	long へのポインタである．それで，単語 long を加える．

最後の例は，内から外への宣言の仕方を説明するために，関数へのポインタをかえす関数の宣言を取りあげている．関数へのポインタをかえす関数を宣言するには，代わりに **double** を返す（つまりさし示される関数は **double** をかえすようにする．

`funct`	名前を書く．
`funct(a, b)`	funct は2つの引数，a，b を持つ関数である．
`*funct(a, b)`	ポインタをかえす関数である．アスタリスクを加える．（ ）は＊より優先順位が高いので，かっこは必要ない．
`(*funct(a,b))( )`	しかし，返り値は関数へのポインタである．ここでは，かっこが必要である．
`double (*funct(a,b))( )`	最後に，funct がかえす返り値にさし示される関数は double である．
	通常，関数の引数の型を宣言する．

```
double (*funct(a,b))( )
int a,b;
{
}
```

7.4　typedef

あまり複雑な宣言は，一般にはよい考えとはいえない．それらは，解釈するのが難しすぎる（そして宣言するのも難しい）．複雑な宣言の代わりに，中間の型をつくる typedef を使うのが好ましい．**typedef** はすでに定義された型にかわって，新しい型を定義するのに使う．例えば，”10 個の **int** の配列” に対する型は

```
typedef int      INTARRAY[10];
```

でつくることができる．構文は単語typedef が前におかれる以外は通常の宣言文と同じである．INTARRAY は実際の型であり

```
INTARRAY    array;
```

と

```
struct
{
        int      key;
        INTARRAY array;
}foo;
```

にあるように，変数宣言に使うことができる．式**sizeof**(INTARRAY) は 20（10 個の **int** の大きさ）に評価される．

INTARRAY 型を他の型をつくるのに使うこともできる．20, 10 要素の配列の配列をつくりたいとき

```
typedef INTARRAY        2D[ 20 ];
```

とすることができる．2D は INTARRAY の 20 要素長の配列であり，各 INTARRAY は 10 要素長の配列である．前に宣言した配列 papa を宣言し直すのに同じ手順を使うことができる（papa は **int** の配列へのポインタの配列へのポインタである）．

```
typedef int          AI [10];     /* 10 個の int の配列              */
typedef AI       *( API [20] );   /* 配列への 20 個のポインタの配列  */
                                  /*                                 */
API                  *papa;       /* 上の配列へのポインタ            */
```

typedef のもうひとつのよい使い方は，複雑な宣言文を持つ関数へのポインタのようなオブジェクトに使う．例えば

```
typedef int     (* PFI)( );
```

は ”**int** をかえす関数へのポインタ” の型をつくる．今後は，”関数へのポインタの配列” を複雑な

```
int     (*functions[20])( );
```

よりも

```
PFI     functions[ 20 ];
```

で宣言することができる.

関数ポインタの配列へのポインタは

```
int     ( *(*pfunct)[ ] )( );
```

ではなく

```
PFI     (*pfunct)[ ];
```

で宣言することができる.

typedef は，特に，キャストを書くときに有効である.

```
(int (*)( )) xp
```

と書くよりも

```
(PFI) xp
```

のほうが理解しやすい.

typedef は次のように，構造体の宣言でも有効である.

```
typedef struct  _tnode
{
        char    *key;
        struct  _tnode  *right ;
        struct  _tnode  *left  ;
} LEAF;
```

LEAF は次のようなことをするのにも使える.

```
LEAF    *root;
LEAF    heap[10];

root    = (LEAF *) malloc( sizeof(LEAF) );
```

7.5 ハードウエアとやりとりするためのポインタの使用

7.5.1 絶対メモリ・アドレッシング

絶対メモリ・アドレスはポインタを通してアクセスされる. 例えば

```
char    *p;
p  = (char *) 0x80;
*p = 10 ;
```

で絶対メモリ番地0x80 に 10 を入れることができる. p は，0x80 に初期化された単なる文字ポインタである. ポインタはアドレスを保持しているのだから，p は

アドレス0x80をさし示している．キャスト［(cher ＊)0x80］は多くのコンパイラで必要である．このキャストは，コンパイラに数0x80(int）を文字ポインタの内容として取り扱うように教える．＊p ＝ 10は10をアドレス0x80に入れる．pは**char**へのポインタであるから，charサイズ分のメモリ（1バイト）だけが変更される．pが**int**ポインタであると宣言されていたら，2バイト変更される．

この例は必要以上に複雑である．pは初期化されてからすぐに使用される．実際には，完全にpなしですますことができる．＊p ＝ 10のpを（char ＊)0x80で置き換えると

```
*( (char *)0x80 ) = 10;
```

になる．再び，これを可能にしたのはキャストである．0x80が文字ポインタの内容として扱うようコンパイラに教えている．

メモリーマップド・ビデオ・ディスプレイのように，もっと複雑なデータ構造は，この方法でアクセスすることができる．例えば，68000上の80カラム×25行のメモリーマップド・ディスプレイは2次元配列のようにみえる．そのディスプレイに対するベース・アドレスを0x10000とすれば，次の文字は番地0x10001に，その次の文字は次の番地に，というように続く．

```
#define addr(row,col) ((char *)0x10000 + (row * 80) + col)
```

で，文字の位置を計算することができる．ここで，ディスプレイの左上の角は(0,0）である．式全体は，文字ポインタに評価される．また，次のように書くこともできるだろう．

```
#define NUMROWS 25
#define NUMCOLS 80

typedef char    DISPLAY[ NUMROWS ][ NUMCOLS ]
#define SCREEN  ( *((DISPLAY  *)0x10000) )

SCREEN[ 2 ][ 3 ] = 'a';
```

ここで，80×25の文字配列に対する**typedef**をつくる．0x10000は，この型の配列へのポインタに型変換（cast）される．それで，(DISPLAY ＊)0x10000は配列へのポインタである，もうひとつのアスタリスクを加えると配列自体（配列ポインタにさし示されるオブジェクト）になり，'[ ]' をつけるとその配列の要素のひとつを参照することができる．コンパイラの中には，ポインタ演算がこんなに複雑になると混乱するものもある．こういうプログラムは有用ではあるが，必ずしも移植性はない．

7.5.2　C プログラムから IBM PC ビデオ・メモリへの直接アクセス

8086 ファミリ CPU では，直接メモリ・アドレッシングは簡単ではない．多くのコンパイラでは，8086 のスモール・モデルで*((char *)0x80) を使用しようとすると，絶対アドレス 0000：0080 にあるセルではなく，現在のデータ・セグメントからオフセット 0x80 にあるセルをアクセスするだろう．データ・セグメントの外にあるセルをアクセスするしかたは複雑である．さらに，この処理はコンパイラによってかなり異なる．

しかし，数をポインタにキャストすれば可能である．再び，上の例を行ってみよう．今度は，モノクローム・アダプタを備えた IBM PC のスクリーンに直接書く．このプログラムは，Microsoft C Compiler（バージョン 3 以上）用である．図 7·21 に示すプログラムは，スクリーンの 4 すみに文字 '1', '2', '3', '4' を書く．その 1, 2, 3, 4, はそれぞれ順にノーマル，アンダーライン，太字，ブリンキング・モードで表示する．

```
#define NORMAL          0x07    /* 基本属性．これらのうちひとつだけ */
#define UNDERLINED      0x01    /* が現れる．                     */
#define REVERSE         0x70

#define BLINKING        0x80    /* 上の属性とお互いに OR される．   */
#define BOLD            0x08

typedef struct
{
        char    letter;
        char    attribute;
}
CHARACTER;

typedef CHARACTER       DISPLAY[25][80];

#define SCREEN  ( *( (DISPLAY far *)0xb0000000 ))

main( )
{
        SCREEN[ 0  ][ 0  ].letter = '1' ;
        SCREEN[ 24 ][ 0  ].letter = '2' ;
        SCREEN[ 0  ][ 79 ].letter = '3' ;
        SCREEN[ 24 ][ 79 ].letter = '4' ;

        SCREEN[ 0  ][ 0  ].attribute = NORMAL              ;
        SCREEN[ 24 ][ 0  ].attribute = UNDERLINED          ;
        SCREEN[ 0  ][ 79 ].attribute = REVERSE | BOLD      ;
        SCREEN[ 24 ][ 79 ].attribute = NORMAL  | BLINKING ;
}
```

図 7・21　IBM PC のメモリーマップド・ディスプレイへの書込み

この問題を解くにあたっていくつか問題がある．第一に，IBM ディスプレイ上の文字は各2バイト必要である．下位バイトが文字で，上位バイトがその属性である．他に影響を与えずに，文字か属性を変更できるようにしたい．それで，2バイトのフィールド，第1のフィールドは文字（letter）で第2フィールドは属性（attribute），を含む構造体CHARACTER の typedef をつくる．8086 は16ビット・ワードの下位バイトを下位のアドレスに格納するから，letter フィールドが構造体の中でさきに宣言されなければならない．次に，DISPLAY という2番目の typedef，CHARACTER の 25×80 配列をつくる．最後に，SCREEN の #define 文をつくる．ビデオ・メモリのベースアドレスを DISPLAY へのポインタにキャストして，そのポインタを通して DISPLAY を参照できるようにもうひとつアスタリスクを加える．この最後の構文は，前の例で使用されているものと同じである．配列へのポインタにさし示されているオブジェクトは配列自体であり，そして '[ ]' の表記をまるで配列名につけるように，そのキャスト定数に適用することができる．

ここで，Microsoft Compiler の特異性が重要な役割をするようになる．予約語 far は，このコンパイラに特有のものである．far は，スモール・モデルのプログラム中に 32 ビットの far ポインタを生成するために使用される．far ポインタは，現在のデータ・セグメント内のデータしかアクセスできない 16 ビットの near ポインタとは違って，メモリのどこの番地でも参照することができる．Microsoft のコンパイラは，far ポインタのセグメント・アドレスを上位16 ビットのワードに，オフセットを下位16 ビットのワードに格納する．IBM のビデオメモリは絶対アドレス 0xb0000 にある．セグメント・アドレスはパラグラフ・アドレスであるので（つまり，絶対アドレスを求めるにはセグメント・アドレスに 16 をかけなければならない），0xb0000 は 8086 にはセグメント：オフセットの形，B000：0000 として表される．far ポインタを初期化するためには，8 けた全部が必要である．それで

```
( *( (DISPLAY far *)0xb0000000 ) )
```

としなければならない．構造体を参照するためにこの式を使っているから，もっとも外側のかっこがここでは必要である．* がドット（.）より優先度が低いから，その式はこの外側のかっこがなければ正しく評価されない．

この準備がすべて整ったら，スクリーンをアクセスするために SCREEN を使

用することができる．SCREEN[0][0] は左上の角，SCREEN[24][0] は左下の角，SCREEN[0][79] は右上の角，SCREEN[24][79] は右下の角である．その文字は

```
SCREEN[row][col].letter
```

で参照される．その属性は次のようにして参照される．

```
SCREEN[row][col].attribute
```

モノクローム・アダプタにサポートされているさまざまな属性が NORMAL（通常），UNDERLINED（下線），REVERSE（反転）として #define されている．さらに，このうちのいくつかが必要に応じて BLINKING（点滅）か BOLD（太字）でビットごとの OR をされている（点滅し（blinking），太字で（bold），下線付き（underlined）の文字等の場合）．残念なことに，下線付きの反転文字はできない（注8）．

7.5.3 モデル・ハードウエアに構造体を使用する

メモリ・マップド I/O システムでは，隣り合ったブロックにあるレジスタ群が制御しているハードウエアは構造体でモデル化できる．SCSI ディスク・コントローラのメモリマップを図7·22 に示す．このコントローラは図7·23 に定義され

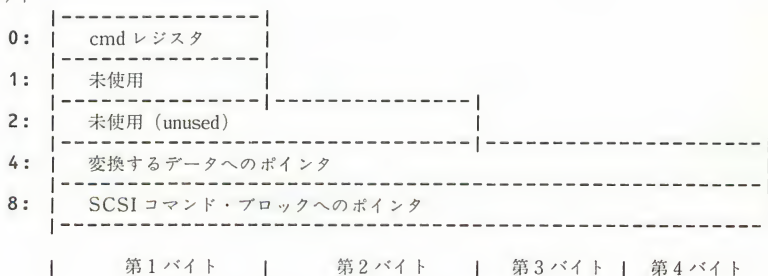



図 7·22 SCSI ホスト・アダプタ

注 8：IBM ディスプレイについての詳しい情報は，Peter Norton の本にのっている．The Peter Norton Programmer's Guide to the IBM PC (bellevue, Wash.：Microsoft Press, 1985) pp. 67-97.

```
typedef char    BYTE;           /* 8ビット      */
typedef short   WORD;           /* 16ビット     */
typedef char    *ADDRESS;       /* 32ビット     */

typedef struct
{
        BYTE    cmdreg;         /* コマンド・レジスタ                  */
        BYTE    unused0;
        WORD    unused1;
        ADDRESS data_block;     /* データ・ブロックのアドレス          */
        ADDRESS cmd_block;      /* SCSIコマンド・ブロックのアドレス */
}
HOST_ADAPTER;

#define DISK    ((HOST_ADAPTER *)0x10000) /* システム・メモリの */
                                          /* ホスト・アダプタの */
                                          /* アドレス           */
```

図 7・23　構造体として定義されたホスト・アダプタ

ている構造体のように表すことができる．

DISK に対する #**define** は最初の例にある (char *)0x80 と同じ機能を果たす．しかし，ここでは定数 0x10000 は HOST_ADAPTER の構造体へのポインタの定数であるかのように扱われる．この定義を使うと

```
DISK->cmdreg = 0xa7;
```

でホスト・アダプタのコマンドレジスタに 0xa7 を入れることができる．-> 演算子を使用しているから，前の例では必要だった * は必要ない．-> 演算子の左はポインタが必要である．データ・ポインタは

```
char    buffer[1024];

DATA->data_block = buffer;
```

で初期化する．

7.5.4　ハードウエア・インタフェースと移植性

移植性は，ハードウエアにインタフェースするすべてのルーチンにとって問題である．第一に，型の大きさが定義できない．**int** はどのくらいの大きさであるか，または int とポインタは同じ大きさなのかがわからない．図 7・23 では BYTE，WORD，ADDRESS の #define はこの問題を最小にしている．移植性の少ない **char**，や **long** 等より，これらの #define がハードウエアの定義に使用される．コンパイラがかわったら，BYTE，WORD，ADDRESS の定義をやり直すこと

ができる．その他の定義はそれに応じて変更されるだろう．

アライメントもまた問題であるかもしれない．このホスト・アダプタは68000 CPU と使用される．16 ビットか 32 ビットのメモリ・アクセスにはワード境界にアライメントをおく．8 ビットのオブジェクトは，偶数か奇数のアドレスでアクセスされる．このハードウエアは，アライメントに注意して設計されている．すべての複数バイトのオブジェクトは，偶数アドレスからおかれる．構造体には unused0 フィールドをおいて，アライメントをきちんと保っている．コンパイラが構造体中のアライメントを保証するので，このフィールドは宣言する必要はない（つまり，構造体でバイトの大きさのフィールドにワードの大きさのフィールドが続くとき，臨時に 1 バイトそこに挿入されることを意味している．いいかえれば，構造体内でフィールドの連続性は保証されない．構造体の大きさは各フィールドの大きさの合計よりも大きいこともある）．はっきりと unused0 を定義すると移植性が備わる．68000 のアライメント制限を持たない計算機（8088 のような）にプログラムを移植することはできる．

最後の問題：大きな数は計算機が違えば格納され方が違う．例えば，68000 は 2 バイト **int** の MSB を下位メモリに格納するが，8086 は下位メモリに LSB を格納する．したがって，ワードの大きさのレジスタを持つハードウエアを動かすソフトウエアは，ハードウエアとソフトウエアを異なる CPU に移植するとき，問題があるかもしれない．ワード内の 2 つのバイトを交換する必要があるかもしれない．このバイトの交換は

```
#define swapb(x) (((x) >> 8) & 0xff) | ((x) << 8)
```

で行うことができるだろう．

7.6 練　　習

7-1　次のプログラムが何をプリントするか説明しなさい．

```
#include <stdio.h>

main( )
{
        static int array[6][2] =
        {
                {' ', 's'}, {'d', 'r'}, {'a', 'w'},
                {'k', 'c'}, {'a', 'b'}, {'c', 'd'}
        };
```

```
		int	*p = (int *)(array + 4);

		for( ++p; p >= (int *)array; putchar( *p-- ) )
			;
	}
```

7-2　次のプログラムが何をプリントするのか？　array は何かことばで説明しなさい．それを図示しなさい．printf( ) への引数全部がどうなるのか，そしてなぜそれらが行うものに評価されるのか説明しなさい．

```
main( )
{
	static char	*array[2][3] =
	{
		{ "  c", "ng ", " no" },
		{ "usi", "fun", "onf" }
	};

	printf("%s%s%s%s\n", **array,   *(*(array+1)+2),
				*array[1], array[0][1]    );
}
```

7-3　次のプログラムが何をプリントするのか？　各行で起こることを説明しなさい．必要なら図を書きなさい．

```
main( )
{
	static char	*p[ ] = { "moldy\n", "jello" };
	static char	**pp  = p ;

	** ++ pp -= 2 ;
	printf("%s ", *pp );

	*(p[0] + 4) = *(*p+3);
	pp[-1][3]   = *(*(pp-1)+2);
	++*--pp ;
	*++*p       = 'r';
	*(*pp - 2)  = **p + 5;

	printf( pp[0] - 2 );
}
```

7-4　次の宣言を書きなさい．

（a）**int** の配列へのポインタの配列へのポインタ

（b）**int** へのポインタをかえす関数へのポインタをかえす関数へのポインタの配列

（c）**float** の配列へのポインタの配列をかえす関数．

（d）各々が **char** へのポインタをかえす複数の関数へのポインタの配列へのポインタをかえす関数

（e） **double** の10要素の配列（へのポインタ）をかえす複数の関数へのポインタの配列へのポインタをかえす関数

7-5　汎用の2分探索ルーチン：

```
bsearch( key, array, nel, width, cmp )
```

を書きなさい．ここで，key は探索することばkeyへのポインタ，array はその配列の最初の要素へのポインタ，nel は配列の要素の数，cmp は比較関数へのポインタである．比較関数は発見したオブジェクトか，key に対応するものが配列内になかったときはNULL をかえす．

7-6　行列は，エンジニアリング・アプリケーションやグラフィックス・アプリケーションで方程式を表すために，簡単形としてよく使用される数の長方形の配列である．それらはコンピュータでは2次元配列として表される．2つの行列は一方の列の数と，他方の行の数が同じならばかけ合わせることができる．行列のかけ算の結果もやはり行列である．その行列は，最初の行列と同じ列の数を持ち，2番目の行列と同じ行の数を持つ．この処理が次の例に表されている．

$$[1\ 2\ 3] \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

は，

$$[(1\times4)+(2\times5)+(3+6)] = [4+10+18] = [32]$$

と評価される．つまり，その積は1×1の行列である．

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 7 & 9 \\ 6 & 8 & 10 \end{bmatrix}$$

の積は3列，2行である．それは次に示されている．

$$\begin{bmatrix} ((1\times5)+(3\times6)) & ((1\times7)+(3\times8)) & ((1\times9)+(3\times10)) \\ ((2\times5)+(5\times6)) & ((2\times7)+(4\times8)) & ((2\times9)+(4\times10)) \end{bmatrix}$$

次のサブルーチンを書きなさい．

```
matrix_mult( prod, a, b, a_row, a_col, b_row, b_col )
char *prod, *a, *b;
int  a_row, a_col, b_row, b_col;
```

a と b は **int** の2次元配列として表される2つの行列の最初のセルへのポインタである．prod は，a と b の積がおかれるメモリ領域へのポインタであ

る．a_row とa_col は，行列 a の行の数と列の数である．b_row とb_col は，行列 b の行の数と列の数である．配列の宣言以外では '[ ]' 表記を使わないで，表記*(p + i) を使用する．すべての配列要素のアクセスはポインタを使用し，ポインタを直接操作することでポインタを更新する．

8　再帰とコンパイラ・デザイン

再帰（recursion）サブルーチンは，自分自身を呼びだすサブルーチンである．多くのプログラミングの本で再帰について述べているけれども，通常は，なぜ再帰サブルーチンのテクニックを誰もが使うのか，理解しにくい単純な例を与えているだけである．再帰は，簡単な動作をより理解しにくくする方法であると思われている．例えば，階乗を計算する場合に，階乗の計算結果をみつけるために再帰を使用することは馬鹿げているだけでなく，メモリを必要もなくむだにしている．しかし，再帰を使ったプログラムの中には，重要なコンパイラや2分木の操作など本当に役立つものもある．このタイプの問題の再帰的解決は，かなりプログラムの量を減らし，同じことを繰りかえし書くよりも理解しやすい．本章では，再帰の妥当な使い方（コンパイラの設計）を調べる．実際のコンピュータ言語の認識から小さな計算式の解析までの問題を減少させることはあきらかな利点である．一方に使われたテクニックは，もう一方に応用可能である．ここでとりあげる式解析プログラムは，計算式を表すASCCI文字列を入力とする．数，かっこ，演算子＋，－，/，＊だけが使用できる．解析プログラムはその評価の結果をかえす．例えば，文字列“(3+1)＊2”をこの解析プログラムに与えると，8がかえされる．このルーチンはちょっと馬鹿げているが，ポイントは複雑な式を解析することではなく，コンパイラを理解することである．コンパイラを調べながら，プログラミング言語を記述する標準表記：バッカス記法（Backus Naur Form, BNF）も考察する．この表記法は，Cコンパイラに使用される構文を説明するのによく使われる．BNFを理解するとC言語の数冊のよりよい参考書を解読するのにかなり役立つだろう．

8.1　再帰はどのように動くか

コンパイラの設計自体に入る前に，実際に再帰がどのように動くのか説明するのが適切であろう．説明するために，さきほどけなした階乗の例を使ってみる（これはとても短いから）．階乗関数は次の式を計算する．

```
N * (N-1) * (N-2) * (N-3) ... * 3 * 2 * 1
```

例えば，5の階乗（通常簡略形で5!）は，

```
5 * 4 * 3 * 2 * 1 = 120
```

である．この式は面白い利点がある．Nの階乗はN＊(N-1階乗)と等しい．具体的な例で調べてみると，式

```
5!
```

は

```
5! == 5 * 4!
```

または

```
5! == 5 * (4 * 3 * 2 * 1)
```

と書き直すことができる．階乗はそれ自体を計算して定義することができる（つまり，階乗の定義は階乗の展開を含むことができる）．それは，問題の再帰的解決を示す，自己参照的な定義のようなものである．しかし，この処理を止める方法を欠いている．特別な場合として0!を定義して処理を止める．

（1）　0!（0の階乗）と1!はどちらも1に等しい．
（2）　それ以外の数に対しては　N!＝N＊(N-1)!

この規則を使うと次のCサブルーチンをつくることができる．それはN!をかえす．

```
factorial( N )
int        N;
{
        if( N <= 1 )                /* 0! == 1! == 1 */
                return 1;
        else
                return( N * factorial(N-1) );
}
```

このサブルーチンがどのように動くか理解するためには，サブルーチンが呼びだされたとき，スタック上に何が起こるか知っておく必要がある（スタック・フレームが何で，それがどのようにつくられるかよくわからなかったら4章を読み直す

FP→	まえのFP	因数のスタック・フレーム
FP+2	リターン・アドレス	
FP+4　N：	3	
		呼出しをしたサブルーチンのスタック・フレーム

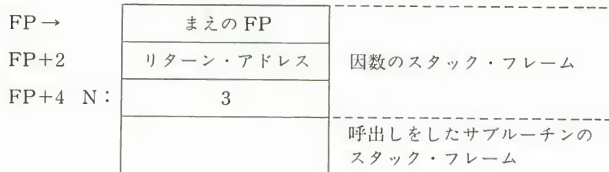



図 8・1　factorial(3) の実行時にできるスタック・フレーム

必要がある)．サブルーチンfactorial(　) が3つの引数とともに呼びだされるとき，図8･1に示されるスタック・フレームがつくられる．

ローカル変数はない．それで，スタック・フレームの中には，factorial(　)に渡されたパラメータN，リターン・アドレス，前のフレーム・ポインタの3つだけがある．変数Nは，いつもフレーム・ポインタからのオフセットを使用してfactorial(　)からアクセスされる．つまり，Nはフレーム・ポインタからオフセット +4(4FP) のところにある．このNは固定メモリアドレスにはない．スタック上にあり，いつもフレーム・ポインタの現在の値からのオフセットを通してアクセスされる．

factorial(　)の中で何が行われるか？　最初に，N＜＝1 が調べられる．それは，Nの値を得るためにフレーム・ポインタからオフセット4にあるメモリ・セルの内容を取り出す．条件に合わなければ，else節が実行される．しかし，そこで計算が行われる前に，サブルーチンfactorial が(再帰的に)呼びだされる．factorial(　)が自分を呼びだしているという事実は，コンパイラの立場からはたいしたことで

FP→	前のFP	2回目の呼出しのスタック・フレーム
	リターン・アドレス	
N：	2	
	前のFP	最初の呼出しのスタック・フレーム(このNは最初の呼出しに属する)
	リターン・アドレス	
N：	3	
		呼出しをしたサブルーチンのスタック・フレーム

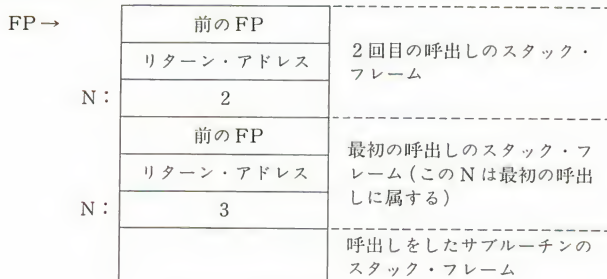



図 8・2　再帰呼出し後のスタック

はない．他のサブルーチンを呼びだすように再帰呼出しをするだけである．スタックにNの値をプッシュして，それからfactorial( )の最初の行に制御を移す．ここで，再帰的に起動されたfactorial( )に渡された値はNではなくN-1である．factorial( )が最初に行うのは，図8·2に示すように，新しいスタック・フレームの設定である．

今，スタック上には2つのNがある．ひとつは最初の呼出しのスタック・フレームにあり，2つ目は再帰呼出しのスタック・フレームにある．しかし，factorial( )の中でNとして使われる値は，計算機が新しい方のフレーム・ポインタからのオフセット4のところに発見するものであるから，2つ目のスタック・フレームにあるNが2番目のfactorial( )の呼出しでは使用される．重大なエラーを除いては，サブルーチンが，他のサブルーチンのスタック・フレームにある変数をアクセスすることはない．それで，factorial( )の再帰呼出しは，もとのfactorial( )のスタック・フレームにあるNを変更することはできない．

もう一度繰りかえすために再帰的な処理が同じ方法で続く．今，3つのスタック・フレームが存在する．この時点のスタックを図8·3に示す．使用されるNの値は，もっとも新しくつくられたスタック・フレームにある．

この時点で，それぞれのスタック・フレームに関係するNの中に，連続するfactorial( )のすべての要素(3,2,1)がスタック上にある．それで後はこれらのNをかけ合わせればよい．サブルーチンは条件N＜＝1を行う．Nがフレーム・ポ

	スタック	
	前のFP	3回目の呼出しのスタック・フレーム(このNは今使用されている)
	リターン・アドレス	
N：	1	
	前のFP	2回目の呼出しのスタック・フレーム
	リターン・アドレス	
N：	2	
	前のFP	最初の呼出しのスタック・フレーム
	リターン・アドレス	
N：	3	
		呼出しをしたサブルーチンのスタック・フレーム

図 8·3　スタックの最後の状態

インタのオフセット4にあるセルから取りだされる．このアドレスにあるセルは1を保持している．それで条件は満たされ，1が呼出しをしたサブルーチンにかえされる．もっとも新しくつくられたスタック・フレームが削除され，図8･2の状態にスタックはもどる．しかし，スタック・フレームが削除される前に，返り値(1)はレジスタにコピーされる．

ひとつ前のfactorial( )の呼出しにもどる．次の行を実行している．

```
return( N * factorial(N-1) );
```

今，呼出しからfactorial( )にかえってきたところである．その呼出しは値1をかえした．その1(レジスタ中にある)にN(スタックにある)をかける．Nのこのコピーは2を保持しているから，2*1すなわち2を呼出したサブルーチンにかえす．現在のスタック・フレームを削除される．factorial( )の呼出しからもどるとき，1を渡すのに使われたものと同じレジスタに返り値(2)をコピーする．

factorial(N-1)からかえってきたこの時点で，factorial( )への最初の呼出しにもどる．返り値をかえすために使われたレジスタは2を含んでいる．スタックは図8･7に示されている状態にもどる．その呼出しからかえされた2にN(フレーム・ポインタからオフセット4にある)をかけ，結果は3*2，すなわち6になる．この6は最初の呼出しをしたルーチンにかえされる．全体の過程を図8･4に図示している．

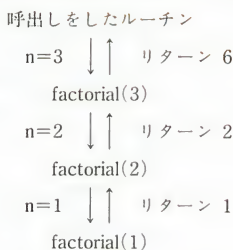



図 8・4　factorial(3)呼出しのサブルーチン・トレース

factorial( )はたくさんのスタックを使う．実際，6バイトのスタック・フレームで，factorial(170)は1Kスタックをオーバーするスタックをつくる(使用できるスタックより多く必要とする)．これは再帰サブルーチンに起こる一般的な問題である．たえずスタックがどんな状態になりつつあるかを考慮し，スタックが大

```
#include <stdio.h>

#define TOO_DEEP  18

long    factorial( N )
long    N;
{
        static   int recursion_level = 0;

        /* 安全な factorial ルーチン．ルーチンが再帰的レベル
         * TOO_DEEP を越えると N！か 0 をかえす
         */

        if( N <= 1 )                  /* 0! == 1! == 1 */
                N = 1 ;
        else
        {
                if( ++recursion_level >= TOO_DEEP )
                        N = 0 ;
                else
                        N = N * factorial(N-1) ;

                --recursion_level;
        }

        return N;
}
```

図 8・5 安全な factorial ルーチン

きくなり過ぎそうならばプログラムで避けるようにしなければならない．スタックの深さを監視する factorial( ) の改良版が図 8･5 に示されている．その改良されたサブルーチンは，再帰レベルの数が TOO_DEEP を越えそうならば 0 をかえす (0!==1 だから通常は 0 はかえされない)．recursion_level は static で宣言されているから，次の再帰レベルで使えるようにその値を得ることができるだろう．static 変数はスタックにはなく，固定アドレスにある．

ここで，スタック・オーバーフローを実際上問題なく行う，別の方法がある．factorial( ) の連続はまったくはやく成長する．実際，12! が符号付き 32 ビット long int で表せるもっとも大きな値である (12! ==479 001 600, 13! ==6 227 020 800, 0x7fffffff==2 147 483 647)．したがって，factorial( ) の先頭で N ＞ 12 か調べ，もしそうならば 0 をかえすべきである．再帰レベル 12 は 96 バイトのスタックしか使わない．それで，N に制限を設ければスタック・オーバーフローを心配する必要はない [スタック・フレームは N が 4 バイトの long，2 バイトのリターン・アドレス，2 バイトの前フレーム・ポインタを想定している (recursion_

level は static だからスタックを使わない)．実際の大きさはコンパイラによって異なる]．

8.2 コンパイラの構造

実用的な，とても小さなコンパイラに応用してみよう．どのコンパイラも3つの機能的に異なる部分を持っている．これらの部分はしばしば組み合わされているが，独立した機能として調べることがもっともよい．コンパイラの最初の部分はトークン認識プログラム(token recognizer)である．トークンとは，入力文字列からのASCII文字をグループ分けしたとき，コンパイラにとって意味のある文字の集合である．つまり，プログラムはトークンの集合としてみることができる．各トークンはひとつ以上連続したASCII文字である．例えば，C言語ではASCII文字；はトークンである．同様に予約語 while もトークンである．そして，＋，＋＋，＋＝などの，演算子のトークンのために複雑になる．トークンはプログラミングの原子のようなもので，分けることができない(wh ile とはいえない)．トークンはいつも内部的には計算上の型か，あるいはファイルのヘッダのどこかにある #define に対応する整数値の集まりとして表される．トークン認識プログラムは，入力列からトークンに対応する整数をかえすサブルーチンである．この例ではトークン認識プログラムは使わない．むしろ，入力を解釈しながら，入力文字列を実際に調べることによって，トークンを認識する．

コンパイラの2番目の部分は，構文解析プログラム(parser)である．これが作業の多くを行う．単語 parse は言語学からきたことばで，言語学でもコンピュータ・サイエンスでも同じ意味を持つ．“(文章を)構成部分に分解し，文法的に記述すること(注1)”である．ただ，文章というところをプログラムに置き換えればよい．コンピュータ言語は，正式な文法で記述される(後で，文法について調べる)．構文解析プログラムはプログラムを構成部分に分解し，文法的な文脈にそって解釈する．すなわち，構文解析プログラムは，トークン認識プログラムからかえされたトークンをコンパイラがコード生成をするのに都合のよい方法に構成する．

処理過程は，構文解析ツリー(parse tree)をつくることができる．例えば，

注 1：The Compact Edition of the Oxford English Dictionary (Oxford：Oxford University Press, 1971) p. 2083.

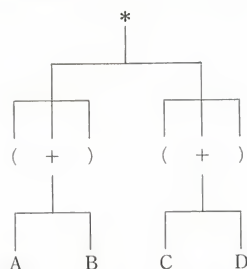



図 8·6　(A+B)*(C+D) の構文要素ツリー

式 (a−b)*(c−d) は，図 8·6 に示されるツリーに構成することができる．

コンパイラの 3 番目の部分，コード生成プログラム (code generator) は構文解析ツリーをある順序で読んで，規則にしたがって命令コードを生成する．例えば，図 8·6 に示すツリーの演算子の後置読みをする（まず，左のノード，中央のノード，右のノード，そしてルートへと繰りかえし進む）．トークンは次の順になっている．

(a b +) (c d +) *

ここで，上の式は，トークンを取りあげながら，ツリーの各トークンに次の規則を適用して計算する．

(1)　トークンがかっこなら，何もしない．

(2)　トークンが変数 (a, b, c, d) なら，その変数をスタックにプッシュする．

(3)　トークンが記号 + ならば，スタックから項を 2 つポップし，それらをたして結果をプッシュする．

(4)　トークンがアスタリスクならば，スタックから項を 2 つポップしてそれらをかけ合わせ，結果をスタックにプッシュする．

構文解析（ツリーの解析）が終わったとき，その答えはスタックの一番上にある．ヒューレット・パッカードの計算機の持ち主は，この処理をよく知っているだろう．

現実のコンピュータでは，適用される実際のルールは，トークンの型と構文解析ツリーでのトークンの位置が互いに関係し作用する．また，多くのコンパイラ（ここで開発しようとしている式解析プログラムを含んでいる）は，構造体やポインタなどで構成するツリーを実際は生成しない．むしろ，構文解析ツリーの構造

は，構文解析プログラムのサブルーチン呼出しシーケンスや，構文解析処理中のプログラムの内部状態に隠されている．

構文解析プログラムには，いくつか選択がある．多くのコンパイラは，表を使った構文解析プログラムを使用している．これは，コンパイラ—コンパイラで自動的につくることができるUNIXユーティリティのYACC（Yet Another Compiler Compiler）は，そんなプログラムの1例である．プログラミング言語の語形記述を与えれば，YACCは汎用の表をもとにした構文解析プログラムが使用できる表をつくる．

多くの公有のコンパイラ（Small Cなど）はあまり洗練された方法は使っていない．これらのコンパイラは，再帰的下向き（recursive descent）構文解析として知られる構文解析の方法を使っている．再帰的下向き構文解析は，表をもとにしたものより理解しやすい．実行もはやく，よりよいエラー・メッセージをだす．一方，それは手で組み立てなければならない．再帰的下向きコンパイラの動作方法をかえるには，コンパイラ自体をかえなければならない．表をもとにしたコンパイラを変更するには，表だけをかえればよい．維持管理が，再帰的下向きコンパイラでは問題になる．

8.3 文法：コンピュータ言語を表現する

プログラムを書き始めるもっともよい方法は，記号形式のようなものに対する問題を減らすことである．アウトライン，流れ図，Warnier-Orr図が主な記号削減の例である．コンパイラもこの処理の例外ではない．コンパイラを書くときは，文法（grammar）という形式的な記号書式で，コンパイルされるプログラミング言語を表すことから始める．どのプログラミング言語も，いくつかの文法で記述することができる．読者が使おうとしている構文解析プログラムのタイプが，どの文法が読者のアプリケーションに適しているかを決定する．文法を記述するために使用するもっとも有効な表記はバッカス記法（Backus Naur Form，略してBNF）である．

式解析プログラムをつくるためには，文法定義から始めなければならない．最初の疑問は，式は正確には何であるか？　ということである．高校時代に，式は因数の組み合わさったものだとおぼえただろう．因数自体（ひとつの数）が式である．演算子で分離された2つの因数も式である．次式は，この規則をBNFで

表したものである．

〈式〉 ：：＝ 〈因子〉
〈式〉 ：：＝ 〈因子〉〈演算子〉〈因子〉

記号：：＝は“として定義する”という意味である．：：＝の左の単語はその定義の名前，：：＝の右は定義本体である．最初の行は，〈式〉が〈因子〉として定義されていると解釈する．行全体はプロダクション（production）と呼ばれる．プロダクション中では論理 OR を表すために従棒（｜）を使用する．

〈式〉 ：：＝ 〈因子〉
｜〈因子〉〈演算子〉〈因子〉

〈式〉の BNF 定義の要素は入力ストリームにみられる実際の記号ではない記号があることに気づくだろう．つまり，〈因子〉と〈演算子〉は実際のプログラムと関係する前に，さらに詳しく定義されていなければならない．〈因子〉のように，さらに定義が必要な記号は非終端記号と呼ばれる．入力にみられる記号は終端記号と呼ばれる．本章では，非終端記号は ’〈 〉’ で囲まれ，終端記号は囲まれていない．演算子になる 4 つの終端記号は ＋，－，/，＊である．演算子の BNF 規則は

〈演算子〉 ：：＝ ＋｜－｜＊｜/

である．

因子を定義するのはちょっと難しい．因子はひとつの数であるが，別の式であってもよい（a＋b－d で，a はひとつ目の因子，b－d は 2 つ目の因子）．因子の BNF 定義は

〈因子〉 ：：＝ 〈数〉｜〈式〉

である．まだ定義されていないのは，〈数〉である．〈数〉は，トークン認識プログ

1）〈式〉 ：：＝ 〈因子〉
｜〈因子〉〈演算子〉〈因子〉

2）〈演算子〉 ：：＝ ＋｜－｜＊｜/

3）〈因子〉 ：：＝ 〈数〉｜〈式〉

4）〈数〉 ：：＝ 集合 ｛0 1 2 3 4 5 6 7 8 9｝
にある ASCII 文字の文字列

図 8・7　簡単な式認識の文法

ラムが発見しやすいので，少しごまかして，ことばで簡単に定義する．ここで使う文法全体を図 8·7 に示す．

：：＝の右にあるどの非終端記号も左側で定義されている．そして，：：＝の左に終端記号はない．例1+2 でこの文法を検査してみよう．1 と 2 はどちらも〈数〉である．だから規則 4 を使って，1 と 2 を同等の非終端記号で置き換えることができる：

1 ＋ 2
〈数〉＋〈数〉

規則 3 によると，ひとつの数は因子でもある．だからまた置き換えを行う：

〈数〉＋〈数〉
〈因子〉＋〈因子〉

＋は規則 2 を使って評価する：

〈因子〉＋〈因子〉
〈因子〉〈演算子〉〈因子〉

最後に，規則 1 を使って，上記をひとつの〈式〉で置き換える：

〈因子〉〈演算子〉〈因子〉
〈式〉

入力のトークン 1+2 を文法の規則を使ってひとつの〈式〉に減らした．だから，1+2 はこの文法にかなった式であるという結論になる．

式にエラーがあったらどうなのか？　1＋＊ を構文解析してみよう．規則 2 と 4 を適用すると

〈数〉〈演算子〉〈演算子〉

になり，さらに規則 3 を適用すると

〈因子〉〈演算子〉〈演算子〉

になる．しかし，これ以上数を減らすために適用できる規則はない．したがって 1－＊ はここでの文法で定義される式ではないという結論になる．

8.4　文法を用いた構文解析

構文解析プログラムは，入力トークンの集合をひとつの非終端記号に減らすプログラムとしてみることができる．式 1+2 の例からわかるだろう．構文解析プログラムを本当のコンパイラにかえるには，コード生成をする他，何か動作をさせ

文法：

（1）〈式〉　　::=　〈因子〉
（2）〈式〉　　::=　〈因子〉〈演算子〉〈因子〉
（3）〈演算子〉::=　+ | − | * | /
（4）〈因子〉　::=　〈数〉
（5）〈因子〉　::=　〈式〉
（6）〈数〉　　::=　'0' から '9' までの範囲にある ASCII 文字の文字列

動作規則：

（1）何もしない
（2）スタックから2つのオブジェクトをポップし，演算子を適用する（規則3にある）．それから結果をプッシュする．
（3）規則2の演算子をおぼえておく
（4）数をスタックにプッシュする
（5）何もしない
（6）ASCII 文字列を数に変換する

図 8・8　文法に動作規則の付加

ることが必要である．それで，動作規則（action rule）を各文法と関連づける．動作規則も加わり少し雑然としているが，ここでの文法を図 8・8 に示す．

文法規則を適用するたびに，等価な動作規則で規定されている動作も行う．1+2 が，図 8・8 の文法で構文解析され，図 8・9 に示されている．もう少し複雑な例を図 8・10 に示す．

たぶん，読者は処理をどのように行うかわかり始めているだろう．動作規則が実際に演算を行うよりも，演算を行うために必要な命令コードを生成することがわかったら，コンパイラを理解していることになる．

残念ながら，今定義した文法はあまり役にたたない．少なくとも，かっこと負数の処理部分を持たせたい．式全体を負に［例えば，−(17*11)］したい．図 8・9 と 8・10 では，式が左から右に構文解析されながら，可能な置換がすべて行われていることに気がつくだろう．もっと現実的な文法はいくぶん複雑な方法で置き換

			規則：	動作：
1	+	2	—	もとの入力文字列
〈数〉	+	2	6	ASCII "1" を int に変換する．
〈因子〉	+	2	4	1 をプッシュする（前のステップの結果）
〈因子〉	〈演算子〉	2	3	+ をおぼえておく
〈因子〉	〈演算子〉	〈数〉	6	ASCII "2" を int に変換する．
〈因子〉	〈演算子〉	〈因子〉	4	2 をプッシュする
	〈式〉		2	1 と 2 をポップし，+ を行い，結果をプッシュする．

図 8・9　図 8・8 の文法を使用しながら 1+2 を式解析する

	規則：	動作：
1+2−3	—	ここから始まる
〈数〉+2−3	6	“1” を int に変換する．
〈因子〉+2−3	4	1 をプッシュする
〈因子〉〈演算子〉2−3	3	＋ をおぼえておく
〈因子〉〈演算子〉〈数〉−3	6	“2” を int に変換する．
〈因子〉〈演算子〉〈因子〉−3	4	2 をプッシュする
〈式〉−3	2	2 つの数をポップし，＋ を行い，結果をプッシュする．
〈因子〉−3	1	何もしない
〈因子〉〈演算子〉3	3	− をおぼえておく
〈因子〉〈演算子〉〈数〉	6	“3” を int に変換する．
〈因子〉〈演算子〉〈因子〉	4	3 をプッシュする
〈式〉	2	2 つの数をスタックからポップし，ひき算を行う，結果をプッシュする．

図 8・10　1+2−3 の式解析

〈式〉	：：＝	〈因子〉	(1)
	|	〈因子〉*〈式〉	(2)
	|	〈因子〉/〈式〉	(3)
	|	〈因子〉+〈式〉	(4)
	|	〈因子〉−〈式〉	(5)
〈因子〉	：：＝	(〈式〉)	(6)
	|	−(〈式〉)	(7)
	|	〈定数〉	(8)
	|	−〈定数〉	(9)
〈定数〉	：：＝	'0' から '9' の範囲の ASCII 文字列	(10)

図 8・11　より現実的な式認識文法

えをする．そして，文法はその複雑さを考慮しなければならない．さらに，文法は，構文解析プログラムが常に現在の入力記号と処理される規則に基づいて，どの規則を適用するのかわかるように構成されていなければならない．よりよい式認識の文法を図 8・11 に示す．実際のプログラムでこの文法を使ってみよう．

8.5　再帰的下向き構文解析プログラム

構文解析プログラムがどのように動いているかみるためにもっともよい方法は，それを調べてみることである．図 8・11 にある文法用の完全な構文解析プログラムを図 8・12 に示す．

構文解析プログラムの適切な討議をする前に，そのプログラムがどのように

```
 1: #include <stdio.h>
 2:
 3: /* EXPR.C:  A small arithmetic expression analyzer.
 4:  *
 5:  *  str にさし示されている式を計算する．かっこがなければ，式は左から右に計算する．有
 6:  *  効な演算子は ＋ ＊ － / であり，それぞれ 加算，乗算，減算，除算に対応する．式は
 7:  *  文字集合 {0123456789＋－＊/()/} で形成されなければならない．空白は認めない．
 8:  *
 9:  *     〈式〉    ：：＝     〈因子〉
10:  *                        |〈因子〉＊〈式〉
11:  *                        |〈因子〉/〈式〉
12:  *                        |〈因子〉＋〈式〉
13:  *                        |〈因子〉－〈式〉
14:  *
15:  *     〈因子〉  ：：＝      (〈式〉)
16:  *                        |－(〈式〉)
17:  *                        |  〈定数〉
18:  *                        |－〈定数〉
19:  *
20:  *     〈定数〉  ：：＝     '0' から '9' の範囲の ASCII 文字列
21:  *
22:  *
23:  *
24:  *------------------------------------------------------------------
25:  *  グローバル変数：
26:  */
27:
28: static  char    *Str ;   /*  構文解析する文字列中の現在の位置．  */
29: static  int     Error;   /*  発見されたエラーの番号              */
30:
31: /*---------------------------------------------------------------*/
32: #ifdef DEBUG
33:
34: main(argc, argv)
35: char    **argv;
36: {
37:         /*  式解析プログラムの練習をするためのルーチン：式がコマンド上に与えられた
38:          *  ら，その式を計算して，結果をプリントする．
39:          *  式がコマンド上に与えられていなければ，式を stdin から (1 行につきひとつ)
40:          *  取り出して計算する．
41:          *  プログラムは構文エラーがあったらシェルに －1 をかえす．インタラクティブ・
42:          *  モードでエラーがあったら，0 をかえす．
43:          *  エラーがなかったら計算結果をかえす．
44:          *
45:          */
46:
47:         char buf[133], *bp = buf ;
48:         int err, rval;
49:
50:         if( argc > 2 )
51:         {
52:                 fprintf(stderr, "Usage:  expr [<expression>]");
53:                 exit( -1 );
54:         }
55:
56:         if( argc > 1 )
57:         {
58:                 rval = parse( argv[1], &err );
59:                 printf(err ? "*** ERROR ***" : "%d", rval );
60:                 exit( rval );
61:         }
62:
63:         printf("Enter expression or <CR> to exit program\n");
64:
65:         while( 1 )
66:         {
67:                 printf("? ");
68:
```

(図 8・12 つづく)

```
 69:                     if( gets(buf) == NULL  ||  !*buf )
 70:                             exit(0);
 71:
 72:                     rval = parse(buf, &err);
 73:
 74:                     if( err )
 75:                             printf("*** ERROR ***\n");
 76:                     else
 77:                             printf("%s = %d\n", buf, rval);
 78:             }
 79: }
 80:
 81: #endif
 82: /*------------------------------------------------------------------*/
 83:
 84: int     parse( expression, err )
 85: char    *expression;
 86: int     *err;
 87: {
 88:        /*  expression（式）の値か，または文字列にエラーがあったときは0をかえす．
 89:         *  ＊Errはエラーの数にセットされる．parseはexpr( )に対するアクセス・ルー
 90:         *  チンである．アクセス・ルーチンを使用すると，expr( )が使うグローバル変
 91:         *  数については知る必要がない．
 92:         */
 93:
 94:        register int    rval;
 95:
 96:        Error = 0;
 97:        Str   = expression;
 98:        rval  = expr( );
 99:        return( (*err = Error) ?  0  :  rval );
100: }
101:
102: /*---------------------------------------------------------------*/
103:
104: static int expr( )
105: {
106:         int     lval;
107:
108:         lval = factor( );
109:
110:         switch (*Str)
111:         {
112:         case '+':  Str++;   lval += expr( );     break;
113:         case '-':  Str++;   lval -= expr( );     break;
114:         case '*':  Str++;   lval *= expr( );     break;
115:         case '/':  Str++;   lval /= expr( );     break;
116:         default :                               break;
117:         }
118:
119:         return( lval );
120: }
121:
122: /*------------------------------------------------------------------*/
123:
124: static int factor( )
125: {
126:         int     rval = 0 , sign = 1 ;
127:
128:         if ( *Str == '-' )
129:         {
130:                 sign = -1 ;
131:                 Str++;
132:         }
133:
134:         if ( *Str != '(' )
135:                 rval = constant( );
136:         else
137:         {
```

（図8・12 つづく）

```
138:                Str++;
139:                rval = expr( );
140:
141:                if ( *Str == ')' )
142:                        Str++;
143:                else
144:                {
145:                        printf("Missing close parenthesis\n");
146:                        Error++  ;
147:                }
148:        }
149:
150:        return (rval * sign);
151: }
152:
153: /*------------------------------------------------------------------*/
154:
155: static int constant( )
156: {
157:        int     rval = 0 ;
158:
159:        if( !isdigit( *Str ))
160:                Error++;
161:
162:        while ( *Str && isdigit(*Str) )
163:        {
164:                rval = (rval * 10) + (*Str - '0') ;
165:                Str++;
166:        }
167:
168:        return( rval );
169: }
```

図 8・12　簡単な式の解析プログラム

構成されているかを述べる．構文解析プログラム中の実際のサブルーチンは何度も繰りかえされる．だから，それらが動作するときにはたくさんのスタックを使用する（これは再帰的サブルーチンには一般的な問題である．再帰的サブルーチンはかなりたくさんのスタックのメモリを使用する）．このスタックの使用量のため，サブルーチンにはできるだけ少ないパラメータを渡すようにしたい（このパラメータがスタックを消費するから）．引数としてサブルーチンに渡される変数をグローバルにすることで，使用するスタックの量を減らすことができる．しかし，これを実際に行うと新しい問題を生みだす．C では，非静的な全グローバル変数はプログラム内の全モジュールに共有される．式解析プログラムはたぶんライブラリ・ルーチンであるだろうし，それにプログラムの残りの通常動作を邪魔してほしくない．さらに，プログラマがグローバル変数ごとに，特定のルーチンで使用されていて，他のところでは使用できないことをおぼえていなければいけないようにはしたくない．この問題は，グローバル変数を static で宣言することで解決する．式解析ルーチン内部で使用するだけのサブルーチンも static にする．ここで，構文解析モジュールの外から静的グローバル変数を初期化する方法が必

要になる．プログラム・リストの 84 行目から始まるアクセス・ルーチン（外部からだけアクセスできるモジュール内のサブルーチン）で初期化を行う．この parse( ) というアクセス・ルーチンは，初期化しか行わない．実際の解析作業をするために expr( ) を呼びだす．

その他の構成上大事なものは，34〜79 行の main( ) ルーチンである．main( ) の第 1 の目的は parse( ) を検査すること—32〜81 行の ♯ifdef/♯endif—である．ライブラリに組み込むためにコンパイルするときには，DEBUG は ♯define しない．main( ) ルーチンはそれだけで適度に役にたつ．コマンド行［expr17/(2＊12)］から式を入力することができる．または，expr とタイプし，簡単な卓上計算機のように，プログラムがプロンプトを出したときに式を入力することもできる．

構文解析プログラムにもどって，図 8･11 のプログラムでみておくことが 2，3 ある．第一に，：：＝ に続くもっとも左の記号は，終端記号かまたは，すべての規則に対して同じ非終端記号である．つまり，〈式〉(expr)に関係する規則はすべて，もっとも左の記号として〈因子〉(factor) を持つ．全〈因子〉のもっとも左の記号は終端記号［(や—］か非終端記号〈定数〉(constant)である．定数のもっとも左の記号は ASCII 数字でなければならない．文法のこの特性は，構文解析プログラムが与えられた状態に適用する規則がどれかを知るために必要である．例えば，〈式〉を評価するときには，構文解析プログラムはまず〈因子〉に関係する規則を適用する．

この文法の第 2 の特性は〈式〉と〈因子〉の定義が再帰的であることである．ひとつの〈式〉が，他の複数の〈式〉の代わりに定義される．〈因子〉の再帰性は 2 段階の深さがある．ひとつの〈因子〉がひとつの〈式〉の代わりに定義され，〈式〉は〈因子〉の代わりに定義される．文法での再帰性は，文法を実現する構文解析プログラムでも再帰を使うことができることを意味する．

適切な文法が与えられたら，文法を直接構文解析プログラムに変換することができる．本章のプログラムでは，文法中のすべての非終端記号が同じ名前の等価なサブルーチンを持つ．〈式〉(expr) に対するサブルーチンは 104 行目から，〈因子〉(factor) に対するサブルーチンは 124 行目から，〈定数〉に対するサブルーチンは 155 行目から始まる．図 8･11 の文法を再びみてみると，すべての〈式〉の規則（1-5）のもっとも左の要素は〈因子〉であることがわかる．同様に，サブルーチン expr( ) の最初は，サブルーチン factor( ) の呼出しである（108 行目）．

文法にもどって，〈式〉が次にすることは終端記号（＊，/，＋，－，ヌル・ストリング）を探すことである．それに対するプログラムは110～117行のswitch文である．デフォルトは終端記号ヌルになる（規則1）．規則2から5の〈式〉の再帰的な評価はswitch文で行われる．

サブルーチンfactor( )はもう少し複雑である．まず，規則7と9の先行するマイナス記号をチェックする(128～132行)．マイナスを取り除くと規則6と7は等しくなる．同様に，規則8と9もマイナスを除くと等しくなる．factor( )は，先行するかっこを探して処理する規則を決定する(134行目)．factor( )がかっこをみつけなかったら，サブルーチンconstant( )を呼びだして規則8が処理される(135行目)．かっこをみつけたら，かっこをとばして規則6が処理され，expr( )が呼びだされる(138～139行)．expr( )がリターンするとき'(' を探してエラー・チェックをすることもできる．

構文解析プログラムの最後の部分は，155～169行のルーチンconstant( )である．このルーチンは基本的にはatoi( )と等しい．しかし，数の直後に文字列ポインタをすすめたり，数がなかったらエラーを示す．

このプログラムには，コンパイラの3つの機能的部分が組み込まれていることに気づくだろう．はっきりしたトークン認識プログラムはなく，むしろ各ルーチンが，処理されるトークンをとび越すようにグローバル文字列ポインタstrを更

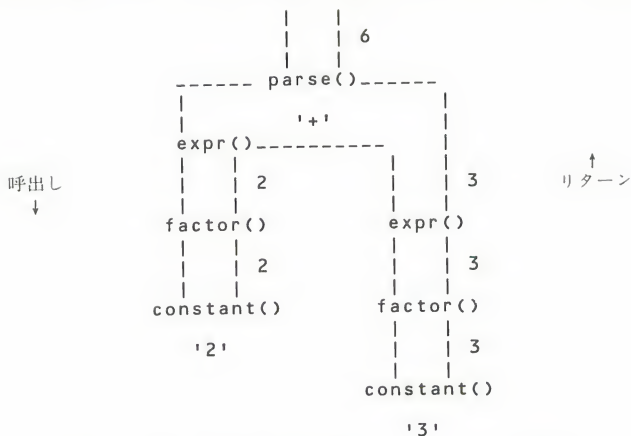



図 8・13　構文解析("2+3", &err)のサブルーチンの軌跡

新する責任を持っている．同様に，コンパイラのコード生成部分も構文解析プログラム自体に組み込まれている（コード生成は，実際にはさまざまなリターン文で肩代わりしている）．

parse( ) 呼出しのサブルーチンの追跡過程を図 8·13 に示す．面白いことに，この図は式 2+3 の構文解析ツリーでもある．構文解析ツリーは本質的に構文解析の過程でつくられる．しかし，この構文解析ツリーはサブルーチン呼出しシーケンスで暗示されている．

8.6 構文解析プログラムの改善

さきの式解析プログラムは再帰のよい例であるが，やはり問題がある．これは，式を左から右に解析するが，実際には式を右から左に計算する（なぜなら，前の再帰レベルでできた結果を使って計算を行うから）．さらに，すべての演算子は同じ優先度で処理されている（本当は + と − は * と / より優先度が低い）．この 2 つの問題は文法を変更することによって除くことができる(注 2)．

演算子の優先度を直すことは簡単である．図 8·11 の文法規則を図 8·14 に示す形に変更するだけである．図 8·14 に新しい終端記号〈項〉が登場している（〈因子〉と〈定数〉の規則は図 8·14 には書いていないが前と同様である）．

演算子の関係は，文法を少しかきまわしたので変更されている．図 8·14 の文法は右再帰である（再帰呼出しがプロダクションのずっと右にある）．右再帰プロダクションは，右から左に関係する．関係を変更するには，図 8·15 にあるように文

```
<式>   ::=   <項>   +  <式>  |  <項>   -  <式>  |  <項>
<項>   ::=   <因子> *  <項>  |  <因子> /  <項>  |  <因子>
```

図 8·14 演算子の優先度つきの式文法

```
<式>   ::=   <式>  +  <項>    |  <式>  -  <項>    |  <項>
<項>   ::=   <項>  *  <因子>  |  <項>  /  <因子>  |  <因子>
```

図 8·15 左から右への結合性をもつ式文法

注 2：ここでは，個々の変換がなぜ行われるのか説明せず，文法的な変換を示している．それらが行うことを忠実に取りさえすればよい．正式な文法とその処理という主題は，コンパイラの設計の本の方がもっと適切である．コンパイラの設計の本を参考文献にいくつかあげている．

```
<式>    ::=    <項> <式'>
<式'>   ::=  + <項> <式'>  |  - <項> <式'>  |  ϵ
<項>    ::=    <因子><項'>
<項'>   ::=  * <因子><項'>  |  / <因子><項'>  |  ϵ
```

図 8・16　式文法の最終形

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3
 4 expr( )
 5 {
 6         return  expr_prime( term( ) );
 7 }
 8
 9 /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
10
11 expr_prime( left )
12 {
13         if( *Str == '+' )
14         {
15                 Str++;
16                 return expr_prime(  left + term( ) );
17         }
18         else if( *Str == '-' )
19         {
20                 Str++;
21                 return expr_prime(  left - term( ) );
22         }
23         else
24                 return left;
25 }
26
27 /*------------------------------------------------------------------*/
28
29 term( )
30 {
31         return  term_prime( factor( ) );
32 }
33
34 /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
35
36 term_prime( left )
37 {
38         if( *Str == '*' )
39         {
40                 Str++;
41                 return term_prime(  left * factor( ) );
42         }
43         else if( *Str == '/' )
44         {
45                 Str++;
46                 return term_prime(  left / factor( ) );
47         }
48         else
49                 return left;
50 }
```

図 8・17　式解析プログラム

法を左再帰にする.

しかし，この最後の変換は，再帰的下向き構文解析には問題である．その文法をできるだけ簡単なプログラムにすると，次のようになる.

```
expr( )
{
        expr( );
        ...
}
```

つまり，expr( )は再帰的に自分自身を呼びだす．永久にその処理を止める方法がないし，スタックはその結果オーバーフローしてしまうだろう．この問題は，図8·16に示すような条件に少し文法を変更すれば解決できる.

これらのプロダクション中で，ϵは，“前にある演算子のいずれかで式が始ま

```
 1 expr( )
 2 {
 3         return  expr_prime( term( ) );
 4 }
 5
 6 /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
 7
 8 expr_prime( left )
 9 {
10         register int    c;
11
12         while( (c = *Str) == '+' || *Str == '-' )
13         {
14              Str++;
15              left = (c == '+') ? left + term( ) : left - term( );
16         }
17
18         return left;
19 }
20 /*----------------------------------------------------------------*/
21
22 term( )
23 {
24         return  term_prime( factor( ) );
25 }
26
27 /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
28
29 term_prime( left )
30 {
31         register int    c;
32
33         while( (c = *Str) == '*' || *Str == '/' )
34         {
35              Str++;
36              left = (c == '*') ? left * factor( ) : left / factor( );
37         }
38
39         return left;
40 }
```

図 8・18 右再帰を除くために変更された式解析プログラム

らなければ，何もしない”ということを意味する．この最終的な文法は図 8・17 に示すサブルーチンに直接変換できる（文法と同様に，サブルーチン factor( ) と constant( ) は同じものであるから，図 8・17 には書いていない）．

これらのサブルーチンには，まだかなり改善の余地がある．expr_prime( ) と term_prime( ) は右再帰である．つまりそれらの再帰的な呼出しは最後に行われる（再帰呼出しとリターンの間には何もない）．したがって，再帰呼出しは while ループで置き換えることができる．条件付の if/else 文で置き換えることもできる．この変更は図 8・18 で行っている．

expr_prime( ) から，すべての再帰呼出しを除くための簡単な変更が他にもある（expr_prime( ) は自分自身を呼びださないで，expr_prime( ) が呼びだすサブルーチンが直接 expr_prime( ) を呼びだす）．expr( ) がすることは expr_prime( ) を呼びだすだけだから，行 3 の expr_prime( ) へのサブルーチン呼出しは expr_prime( ) 自体で置き換えることができる．term( ) と term_prime( ) にも同じ

```
 1 expr( )
 2 {
 3         register int    c;
 4         register int    left;
 5
 6         left = term( );
 7
 8         while( (c = *Str) == '+' || *Str == '-' )
 9         {
10             Str++;
11             left = (c == '+') ? left + term( ) : left - term( );
12         }
13
14         return left;
15 }
16
17 /*-------------------------------------------------------------*/
18
19 term( )
20 {
21         register int    c;
22         register int    left;
23
24         left = factor( );
25
26         while( (c = *Str) == '*' || *Str == '/' )
27         {
28             Str++;
29             left = (c == '*') ? left * factor( ) : left / factor( );
30         }
31
32         return left;
33 }
```

図 8・19　expr( ) と term( ) の最終版

ことがいえる．これらの変更を行った expr( ) と term( ) の最終プログラムは図 8・19 に示されている．

8.7 練　　習

8-1　次の再帰サブルーチンを書きなさい．

```
search( array,  key, array_size )
int    *array,  key, array_size;
```

これは，そこに示されている整数配列に key を探す 2 分探索を行う．array_size は配列の要素の数で，バイト数ではない．このルーチンは key が発見されたセルへのポインタをかえす（また，key が発見されなかったら −1 をかえす）．

8-2　自分のスタック・フレームの大きさを，バイト数でプリントする C 言語のサブルーチンを書きなさい．この大きさは実行時に計算すること（サブルーチンを 10〜15 行より大きなプログラムにすると，たいへん悪いことになるだろう）．

3-3　練習 8-2 の原則を使用して，式解析プログラムが式全体の解析に使用するスタックの最大量を，parse( ) がプリントするように式解析プログラムを変更しなさい（つまり，再帰のもっとも深いレベルで使用可能なすべてのスタック・フレームの大きさの和をプリントする）．parse( ) 自体に使用されているスタック・フレームの大きさがこの数に含まれているはずである．“((1∗2)＋(3∗4))” の計算にどのくらいのスタックが使用されるか？

8-4　本章で説明した構文解析プログラムを使用して，式 “1＋(2∗3)” を解析し，各サブルーチンの呼出しの後でそれらのスタックとサブルーチンの追跡図を示しなさい．すべての変数とアドレスが各々 2 バイト必要だとすると，この式を計算するのにどのくらいのスタック・メモリを必要とするのか？　どのくらいスタックの必要量を減らすことができるか？　再帰レベルを制限するためにどのように parse( ) を変更することができるか？

8-5　標準入力から対話的に文を入力し，その文を評価する卓上計算機プログラムを書きなさい．その計算機は，任意の英文字名を持つ任意の数の変数をサポートしていなければならない．文 “〈name〉＝〈expr〉;” が変数を生成し（まだその変数が存在していなければ），〈expr〉の値をその変数に割り当

てる．その後，その変数は式中で数の代わりに使うことができる．print 文（“print <expr>;”）は式の計算の結果を標準出力にプリントする．次の文法を使用する．

```
<stmt>  ::= print <expr> ; <stmt> | <name> = <expr> ; <stmt> | ε
<expr>  ::=   <term> <expr'>
<expr'> ::= + <term> <expr'> | - <term> <expr'> | ε
<term>  ::=   <fact> <term'>
<term'> ::= * <fact> <term'> | / <fact> <term'> | ε
<fact>  ::= ( <expr> )       | -( <expr> )      | <const> | -<const>
<const> ::= <num> | <name>
<num>   ::= '0' から '9' の範囲の ASCII 文字の文字列
<name>  ::= 'a' から 'z' の範囲の ASCII 文字の文字列
```

文はすべてセミコロンで終わる．エラー（セミコロンや等号がないなど）があったときは適切な対応を行いなさい．

9 Printf( )の構造

本章では，C 言語の特徴全体を使用するサブルーチンである printf( ) のインプリメンテーションを書くことによって，前の章で取りあげた点をいくつか応用する．printf( ) は，洗練されたやり方でポインタを使用している．不定数の引数を持っている．そして，キャスト，型変換などをよく理解している必要がある．printf( ) は，現実的なプログラミングの例でもある．printf( ) を書いたときに多くの妥協をしており，多くの同様な妥協が，現実のプログラムを書くときにも行われる（参考書のプログラムと比較すると）．

9.1 スタックの再考

printf( ) にとり組むまえに，問題の範囲を少し減らしておこう．次のサブルーチンは，自分のリターン・アドレスをプリントする．

```
typedef int     (*PFI)( )

printaddr( arg )
{
        PFI          *argp = &arg ;

        printf( "printaddr: called from 0x%04x\n", argp[-1] );
}
```

printaddr( ) がどのように動くかみるために，スタック・フレームを調べてみよう．

```
          |                 |   アドレス   <-SP
          |-----------------|
  argp:   |       106       |   100
          |-----------------|
          | 前フレーム・ポインタ |   102      <-FP
          |-----------------|
          |  リターン・アドレス  |   104
          |-----------------|
   arg:   |     パラメータ     |   106
          |-----------------|
          |                 |
```

式 &arg は arg のアドレス，106 を計算する．したがって，argp＝&arg は 106 を argp に入れる．argp はポインタだから，argp [0] はアドレス 106 にあるセルの内容である．argp はポインタの大きさのオブジェクトへのポインタだから，argp [−1] は argp の内容からポインタの大きさをひいたアドレスにあるセルをアクセスする．この計算機ではポインタは 2 バイト必要とし，argp は 106 を含んでいる．それで，argp [−1] は 106−2 (104 番地) にあるセルを参照する．セル 104 は，サブルーチンのリターン・アドレスを保持している．

このルーチンは移植性はない．しかし，たいていのコンピュータで動作する．スタック・フレームがある方法で構成されるとする．実際はコンピュータによってスタック・フレームの構造は少し異なる．しかし，リターン・アドレスと引数の相対位置はたいてい同じである．printaddr( ) もサブルーチンへのポインタとサブルーチンのリターン・アドレスが同じ大きさであると想定している．これはよい考えであるが，必ずしもそうではない．しかし，printaddr( ) は，C プログラムがどのように直接スタックをアクセスするのか説明するのに役立つ．printf( ) がどのように動作するのか理解するために，printaddr( ) の処理を理解する必要がある．

9.2　printf( )

printf( ) と fprintf( ) のソース・プログラムを図 9·1 に示す．extern 文の構文をよく知らないかもしれない．それは審議中の ANSI 規格に規定されている構文で，サブルーチンの引数の型チェックを強力に行う．その文をさきの規格ではファンクション・プロトタイプ (function prototype) と呼ぶ．(今のところは Microsoft と Lattice のコンパイラでサポートされている．他のコンパイラ・メーカーは始めたばかりである)．宣言

```
extern  void   doprnt ( int (*)( ), int, char*, char** );
```

```
 1   #include <stdio.h>
 2
 3   extern  void    doprnt ( int (*)( ), int, char*, char** );
 4   extern  void    fputc  ( int, FILE * );
 5
 6   printf( format, args )
 7   char    *format, *args;
 8   {
 9           doprnt( fputc, stdout, format, &args );
10   }
11
12   fprintf( stream, format, args )
13   FILE    *stream;
14   char    *format, *args;
15   {
16           doprnt( fputc, stream, format, &args );
17   }
```

図 9・1 printf( ) と fprintf( )

は，doprnt( )が値をかえさない(それがvoidである)サブルーチンであるといっている．これは4つの引数をとる．最初はintをかえすサブルーチンへのポインタである．2番目はint，3番目はcharへのポインタ，そして，4番目はcharポインタへのポインタである．型の並びにある構文は，キャストに対する構文のようである（ただし，キャストの外側のかっこは除く）．コンパイラはこの引数並びに適合しない型や，引数の個数の間違い，間違って使用された返り値をみつけたら，エラー・メッセージをプリントする（通常は重大なエラーではなく，警告をだす）．可能ならば，コンパイラはサブルーチンをうまく呼びだすために必要な型変換を行う．不定数の引数を持つサブルーチンもまた規定することができる．

例えば

```
extern void     printf(char *, ... );
```

型並びの省略符号が，最初のcharポインタの引数にどんな型の引数がいくつ続いてもよいことを表している（注1）．

みてわかるように，printf( )は5行の長さしかない．printf( )は仕事の大半を行うために，便利に使える関数doprnt( )を呼びだす．fprintf( )も同じ関数を呼びだす．それで，doprnt( )は，いくつか特別な関数を書くよりも，汎用の便利な関数をひとつ書いて，プログラムの大きさを小さくするというよい例である．

doprnt( )に話を移す前に，doprnt( )に渡される引数を調べてみよう．

注 1：ANSI規格の初期の案では省略符号を使わず，カンマのみであった．

doprnt( ) への最初の引数は，出力関数へのポインタである．その関数は

```
(* out)( c, o_param );
```

で間接的に呼びだされる．ここで，c は出力する文字であり，o_param は doprnt( ) への 2 番目の引数である．つまり，o_param は fputs( ) へ引数列を渡すための手段を与える．doprnt( ) 自体は o_param を使用しない．doprnt( ) の 3 番目の引数は，printf( ) に渡されたのと同じフォーマットの文字列へのポインタである．4 番目はその他の引数へのポインタである．いいかえれば，4 番目の引数はスタック上にある printf( ) への引数のアドレスである．そのアドレスにすぐフォーマットを書いてある文字列が続く．その状態を図 9·2 に示す．

```
呼出し               x = 4; y = 3;
                     printf( "%d %d", x, y );
は
                     doprnt( fputc, stdout, format, &x );
を呼びだす.
次のようなスタックフレームが生成される.

              |                  |
              |------------------|  - - - - - - - - - - - - - - - - - -
              |                  |           doprnt のスタック・フレーム
              | doprnt のローカル |
              | 引数             |
              |                  |  100
              |------------------|
         FP-> | 前フレーム・ポインタ|  102
              |------------------|
              | リターン・アドレス |  104
              |------------------|
        out:  | fputc のアドレス  |  106
              |------------------|
    o_param:  |     stdout       |  108
              |------------------|
     format:  |   80   *---------|----------+
              |------------------|          |
+------ ap:   |    120           |  112     |
|             |------------------|  - - - - - - - - - - - - - - - - - -
|        FP-> | 前フレーム・ポインタ|  114     |      printf のスタック・フレーム
|             |------------------|          |
|             | リターン・アドレス |  116     |    80: 81: 82: 83: 84: 85:
|             |------------------|          +---> |-----------------------------|
|    format:  |  80    *---------|---------------> |'%'|'d'|' '|'%'|'d'|'\0'|
|             |------------------|                |-----------------------------|
+--> args:    |    4             |  120
              |------------------|
              |    3             |  122
              |------------------|  - - - - - - - - - - - - - - - - - -
         x:   |    4             |  124   呼出しをしたルーチンのスタック・フレーム
              |------------------|
         y:   |    3             |  126
              |------------------|
        FP->  |                  |  128
```

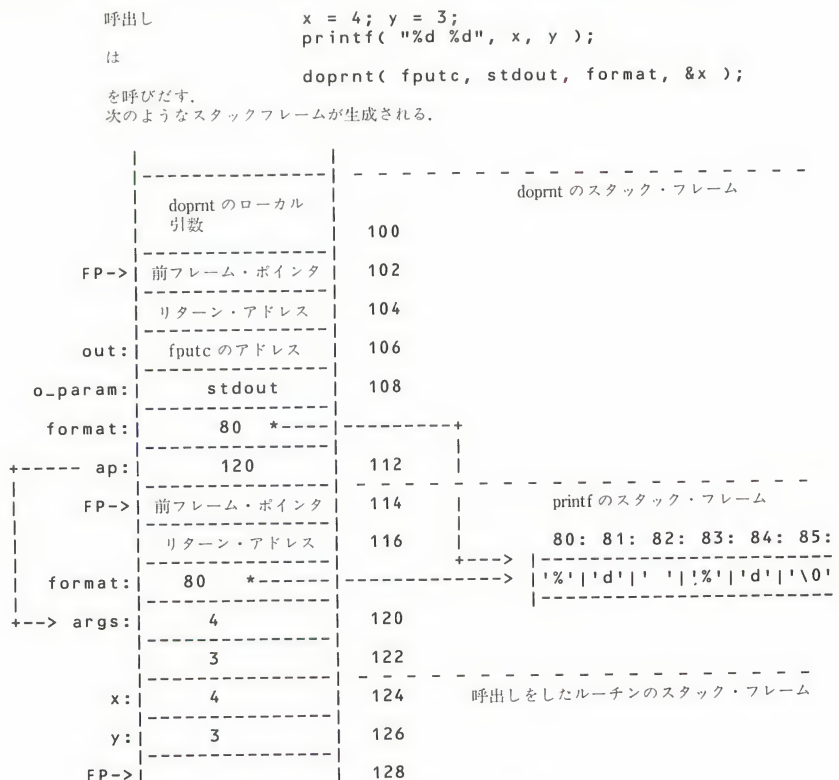



図 9・2　doprnt( ) の引数

printf( ) が呼びだされるとき，3つの引数は逆順にスタックにプッシュされる．そのため変数yの内容は122番地にプッシュされ，xの内容は120番地にプッシュされる．フォーマットの文字列へのポインタは，118番地にプッシュされる．リターン・アドレスは，サブルーチン呼出しの一部としてプッシュされ，それからprintf( ) が前のフレーム・ポインタをプッシュする．printf( ) はローカル変数を持たないから，スタック・フレームはそれで終わりである．

printf( ) がdoprnt( ) を呼びだし，doprnt( ) への引数が呼出し処理の一部として逆順にスタックにプッシュされる．はじめにfputc( ) へのポインタ（fputc( ) サブルーチンの最初の命令のアドレス）がまずプッシュされる．fputc( ) への2番目の引数はstdoutである．次にstdoutが，スタックにプッシュされる．そして，printf( ) へのフォーマット引数の内容が次にプッシュされる．そのため，スタック上には，フォーマットの文字列への2つの同一のポインタが存在する．ひとつはprintf( ) のスタック・フレーム上に，もうひとつはdoprnt( ) のスタック・フレーム上にある．最後に，argsのアドレスがプッシュされる．

argsは，実際にいくつの引数がprintf( ) に渡されたかということには関係なく，いつも同じprintf( ) スタック・フレームの相対位置にある．つまり，argsはいつもフレーム・ポインタから同じオフセットのところにある．いくつかの引数が，スタック上のargsに続く（より上位のアドレスにある）．サブルーチンの引数は逆順にプッシュされるため，最初の引数が，スタック・フレームのどこに位置しているか必ずわかる．このことを知らないで，不定数の引数を持つサブルーチンを書くことはできない．

9.3　doprnt( ) で使用されるマクロ

doprnt( ) の頭の部分を図9·3に示す．

2つの外部ルーチンltos( ) とdtos( ) が使われている（13，14行で宣言されている）．その2つについて簡単に述べる．

2つのサブルーチンには＃defineのようにextern文が続く（26～30行）．doprnt( ) を書くときに行った取引のひとつは，実行速度対プログラムの大きさだった．プログラムの実行時間の多くはたいていI/O動作を行うために費やされるから，速度を考えてprintf( ) を最適化するのが合理的なように思われる．したがって，サブルーチンを実行するためには余分な時間がかかるから，doprnt( ) のサブルーチ

```
 1 /* DOPRNT / FDOPRNT
 2  *
 3  *        printf, sprintf 等の実際のフォーマッティング・ルーチン
 4  *
 5  *        —%f 変換をとサポートするために FLOAT を #define する（これはコマン
 6  *            ドの w/ a -dFOLAT オプションによって行われる．この場合，
 7  *            コンパイルされたサブルーチンは doprnt( ) より fdoprnt( ) を呼び
 8  *            だす．
 9  *
10  *        —標準ではない変換 %b（2進）変換をサポートしている．
11  */
12
13 extern char      *ltos(long,    char*, int);
14 extern char      *dtos(double, char*, int);
15
16 /*-----------------------------------------------------------
17  *      下の命令コードのメモリを節約するためのマクロ．コンパイラが複数のマクロ
18  *      を許容しないならば，逆スラッシュを削除して，定義全体を1行にまとめるこ
19  *      と．
20  *
21  *        PAD(fw)      filchar を fw 回出力する．fw=0.
22  *        TOINT(p,x)  p が数を越えるところに更新される以外は
23  *                     x=atoi(p)；のように動く．
24  */
25
26 #define PAD(fw)       while( --fw >= 0 )                      \
27                              (*out)( filchar, o_param )
28
29 #define TOINT(p,x)   while( '0' <= *p  &&  *p <= '9' )     \
30                              x = (x * 10) + (*p++ - '0')
31
32 /*-----------------------------------------------------------
33  *        INTMASK は long の下位 N ビットをマスクして移植性を持たせる方法であ
34  *        る．ここで，N は int のビット幅．
35  */
36
37 #define INTMASK              (long)( (unsigned)(~0) )
38
39 /*-----------------------------------------------------------
40  *  多くのコンパイラは次の式を受け入れない．
41  *
42  *                              int     *ap;
43  *                              x = *( (long *)ap )++;
44  *
45  *   したがって，次の形を使用してきた．
46  *
47  *                             char *ap;
48  *                             x = *( (long *)ap );
49  *                             ap += sizeof(long);
50  *
51  *   あまりきれいではないが，このほうが移植性がある．
52  */
53
54 #define NEXTARG(x, type)                           \
55         {                                          \
56                 x = *(type *)ap;                   \
57                 ap += sizeof(type);                \
58         }
```

（図 9·3 つづく）

```
59
60 /*-----------------------------------------------------------
61  *      FLOAT が #define されていれば，サブルーチン名として fdoprnt が使用さ
62  *      れる．#define されていなければ，doprnt が使用される．
63  */
64
65 #ifdef  FLOAT
66 #define DOPRNT    fdoprnt
67 #else
68 #define DOPRNT    doprnt
69 #endif
70
```

図 9・3 doprnt.c：#define 等

ン呼出しの回数を少なくする．いっぽうで，サブルーチンの構成（サブルーチンを独立した単位にし，呼びだすルーチンとは分離する）がプログラムをより維持管理しやすくする．マクロのようなサブルーチンを使用することをおすすめする．しかし，これらのマクロにはそれの波及効果などには注意しなければならない．

例えば，(誤った) マクロの呼出し

```
TOINT( p++, x );
```

は，次のように展開される．

```
while( '0' <= *p++  &&  *p++ <= '9' )    \
        x = (x * 10) + (*p++++ - '0')
```

p の最初の 2 つの展開は，p を 2 回インクリメントするという望ましくない結果になった．3 番目の展開 p＋＋＋＋ は規則違反である．その他にも問題がある．TOINT が正しく使用されていたとしても，p は走査するけたを通り過ぎてしまうだろう．サブルーチンはけたを通り過ぎることはない(少なくとも直接には)．それでサブルーチンが正しく使用されたときでさえ，マクロは波及効果を及ぼす．もうひとつ考えなければならないことは，マクロを次の行に続けるために使う逆スラッシュである．プログラムは，きちんとフォーマットされていれば読みやすい．しかし，コンパイラの中には複数行のマクロをサポートしていないものもある．このようなコンパイラでは，定義を 1 行にまとめなければならない．

次の #define は行 37 にある．

```
#define INTMASK   (long)( (unsigned)(~0) )
```

INTMASK は long を等価な int 切り捨てるために使用される．つまり，32 ビットの long を 16 ビットの int にする．

```
longvar &= INTMASK;
```

は数の上位16ビットをクリア(0にセット)する．式

```
#define INTMASK    0xffff
```

は動作しない．ここで問題になるのは自動型変換である．0xffffはintである．そしてそれは負数(−1)である．式がlongとintの両方を含んでいるときには，式が計算される前にintがlongに変換される．0xffffは負数だから，0xffffffffに変換される(符号拡張のため)．だから，longvar &=0xffffはコンパイラにはlongvar &=0xffffffffとして扱われ，その式は何もしないだろう．この問題は次のようにして避けることができる．

```
#define INTMASK    0xffffL
```

最後のLは，コンパイラに0xffffが32ビットのlongで，下位16ビットを1にセットするように知らせている．しかし，このプログラムは移植性がない．intは16であると想定している．式：

```
#define INTMASK    (long)( (unsigned)(~0) )
```

は移植性がある．intの大きさを想定していない．ここで，~0は全ビットが1にセットされているintの大きさのオブジェクトである．この数がlongで式中に使用されているとき，符号拡張を避けるために，unsignedにキャストする．最後にこの数をlongにキャストする．

しばらくの間，NEXTARGの＃defineを調べてみよう．65〜69行にある次の＃defineは，最後のサブルーチンの名前を決定する．FLOATが＃defineされていたら，そのときはサブルーチン名はfdoprnt( )である．そうでなければ，doprnt( )である．同じソース・プログラムで，このようにひとつのサブルーチンから2つの変形版を生成するために使用することができる．ひとつは浮動小数点をサポートするfdoprnt( )を呼び，もうひとつはサポートしないdoprnt( )を呼ぶ．

9.4　スタックからの引数のフェッチ

もうすでにdoprnt( )への不定数の引数について述べてきた．ap，(スタック上の) その引数へのポインタ，は文字ポインタとして宣言されている．これは，コンパイル時には引数の型はわからないので，必要である．フォーマット文字列を解析して，実行時にこれらの型を推定しなければならない(%dと%cはintの大きさの引数に対応し，%fはdoubleの大きさの引数に対応する等)．apをひとつの引数から他の引数にインクリメントさせるとき，apにはさし示されるオ

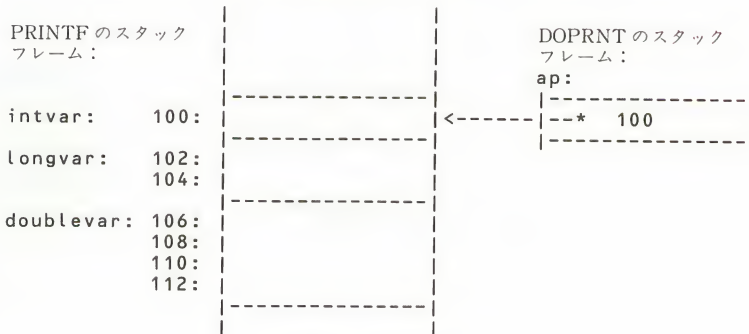



図 9・4 引数ポインタの使用

ブジェクトの，スタック上のひとつの引数の，大きさを加えなければならない．apをcharポインタにすることで，このインクリメントは簡単に行うことができる．charポインタのポインタ計算はただの計算である．charポインタに1を加えることは，実際にそのポインタの内容に1を加える（なぜなら．sizsof(char)＝＝1だから）．

apがdoprnt(　)でどのように使用されているかみてみよう．printf(　)の呼出しとその結果のスタック・フレームを図9·4に示す．ここで，apはprintfにあるフォーマット文字列（&args）が続く最初の引数をさし示すように初期化される．apにさし示されているオブジェクトをフェッチするためには，次のようにしなければならない．

```
int     x;

x = *( (int *)ap );
```

apはcharポインタだからキャストは必要である．apはcharへのポインタというより，むしろintへのポインタであるかのように扱われる．式

```
x = *ap;
```

はintで使用される2バイトでなく，1バイトをフェッチするだろう．式x＝

＊((int ＊)ap); は ap にさし示されるオブジェクトをフェッチする．しかし，そのオブジェクトは，char の大きさでなく int の大きさである．だから int 全体をフェッチする．

ap を，その int を過ぎて次の引数をさすようにインクリメントするには，ap に int の大きさを加えなければならない．

```
ap += sizeof( int );
```

ap は char ポインタであるから，ap には 2 を加える．この新しい状態を図 9·5 に示す．

```
x = *( (int *)ap );
ap += sizeof(int);
```

次のように行う：

```
                       |                   |
PRINTFのスタック        |                   |          DOPRNTのスタック
フレーム：              |                   |          フレーム：
                       |                   |
                       |-------------------|
intvar:        100:    |         3         |          ap:
                       |-------------------|          |----------------|
longvar:       102:    |                   |<-------- |--*   102       |
               104:    |                   |          |----------------|
                       |-------------------|
doublevar:     106:    |                   |
               108:    |                   |          x:
               110:    |                   |          |----------------|
               112:    |                   |          |       3        |
                       |-------------------|          |----------------|
                       |                   |
```

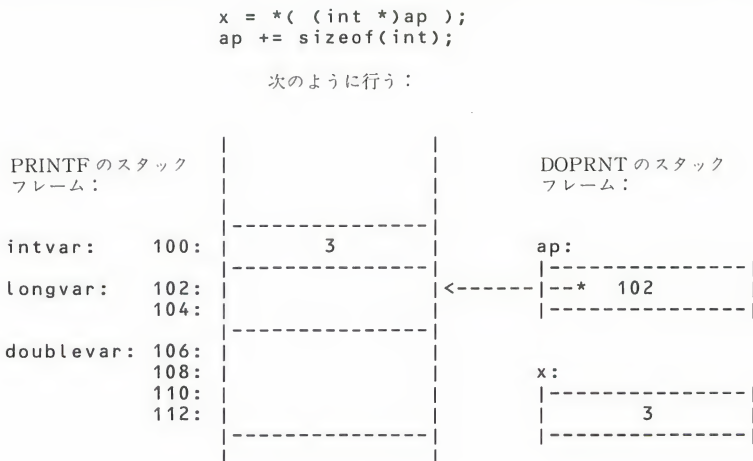



図 9・5　引数ポインタの更新

ap は，今スタックの 2 番目の引数 longvar をさし示している．前と同じようにしてこの値をフェッチすることができる．しかし，異なるキャストを用いる．

```
long    lx;

lx = *( (long *)ap );
ap += sizeof( long );
```

は，ap にさし示されている long の大きさのオブジェクトを lx に入れる．そして，ap に 4 (long の大きさ) をたして，ap が double をさし示すようにする．この新しい状態を図 9·6 に示す．

```
x = *( (long *)ap );
ap += sizeof(long);
```

は次のように行う：

```
                          |                |
PRINTF のスタック         |                |          DOPRNT のスタック
フレーム：                |                |          フレーム：
                          |                |
                          |----------------|
intvar:        100:       |       3        |          ap:
                          |----------------|          |--------------|
longvar:       102:       |     70000      |    +---|--*   106      |
               104:       |                |    |     |--------------|
                          |----------------|    |
doublevar:     106:       |                |<---+
               108:       |                |
               110:       |                |          lx:
               112:       |                |          |-------------------------|
                          |----------------|          |          70000          |
                          |                |          |-------------------------|
```

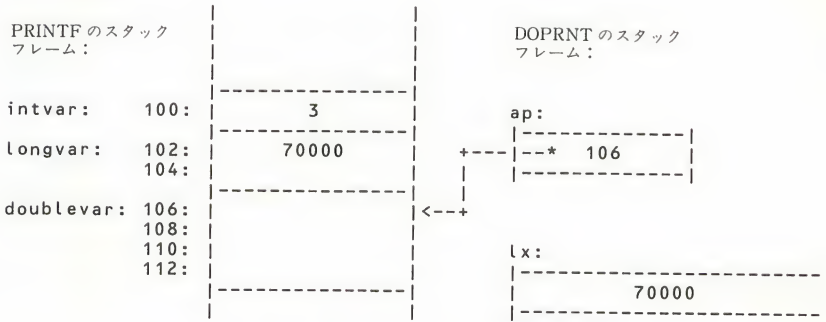



図 9・6 引数ポインタを再度更新する

double は

```
double  dx;

dx = *( (double *)ap );
ap += sizeof( double );
```

でフェッチすることができる．NEXTARG(x, type) マクロは，示された型の引数をフェッチして，それを x に入れる．それから，その引数を過ぎるようにap をすすめる．NEXTARG (lnum, long) は次のように展開される．

```
{
        lnum = *(long *)ap;
        ap += sizeof(long);
}
```

NEXTARG の本体は '{ }' で囲まれている（なぜなら，2つの文からできているから）．次のように書くことができる．

```
if( condition )
        NEXTARG( x, int );
```

しかし，最後の else は '}' のために問題が起こる．ここではセミコロンは if に続くヌル文として解釈される．この問題を訂正するためには，セミコロンのひとつを削除する．

```
if( condition )
        NEXTARG( x, int )          /* No ; */
else
        NEXTARG( x, int );
```

このようなことをするときには，セミコロンがないことがエラーにみえないように，プログラムに必ずコメントをつける必要がある．

ひとつのマクロに2つの文がある問題を処理する他の方法は，コンマ演算子を使うことである．次のようにNEXTARGを#defineすることができる．

```
#define NEXTARG(x, type) x=*(type *)ap, ap += sizeof(type)
```

ここで，コンマ演算子の左から右の評価を行う．この式は読みにくい（少なくとも筆者にとっては）．

NEXTARGには他に2つの方法がある．

```
#define NEXTARG(x,type)  x = *( (type *) p )++
```

はスタックから引数を取り出して，ポインタをひとつの動作で更新する．しかし，多くのコンパイラはこの文を受け入れない．問題は，コンパイラが使用する一時的な変数rvalueを処理しなければならないことである．キャストされた数の値は，しばしばrvalueに入れられる．つまり，キャストは変換されるオブジェクトの大きさをかえるから，このオブジェクトはしばしば名前のわからない一時変数に入れられる．この場合，上の式はコンパイラに，p自体ではなく，一時変数rvalueをインクリメントするように教えている．rvalueをインクリメントしても意味がないから，多くのコンパイラはこの式を拒絶する．

NEXTARGのもうひとつの方法は

```
#define NEXTARG(x,type)  x = ((type *)( p += sizeof(type) ))[-1]
```

である．ここで，pはcharへのポインタである．pは＋＝を使って取り出したいオブジェクトを越したところに更新される．[−1]で新しい位置からの指標づけをして，スタックを逆にもどる．キャスト演算子は＋＝よりも優先度が高く，かつ[ ]よりも優先度が低いので，この式のすべてのカッコが必要である．

9.5　doprnt( )：プログラム

doprnt( )の実際のプログラムが図9·7に示されている．プリントが簡単な場合，つまり変換を行う必要がない場合は，94〜99行のプログラムですべてが行われる．一度に1文字ずつフォーマットの文字列を走査して，直接出力ルーチンを

```
 71 /*----------------------------------------------------------------*/
 72 
 73 char *DOPRNT (out, o_param, format, ap)
 74 int      (*out)( );              /* 出力サブルーチン                    */
 75 int      o_param;               /*  out( )に渡す第2引数               */
 76 char     *format;               /*  フォーマット文字列へのポインタ     */
 77 char     *ap;                   /*  引数へのポインタ                   */
 78 {
 79          char    filchar ;    /*  欄を埋めるために使用される文字     */
 80          char    nbuf[34];    /*  変換された文字列を保持するバッファ */
 81          char    *bp     ;    /*  現在の出力バッファへのポインタ     */
 82          int     slen    ;    /*  bpにさし示される文字列の長さ       */
 83          int     base    ;    /*  現在の基数 (%x=16, %d=10, 等)      */
 84          int     fldwth  ;    /*  %10xにあるようなフィールドの幅     */
 85          int     prec    ;    /*  %10.10xか%10.3fでの精度            */
 86          int     lftjust ;    /*  1=左をそろえる. (例, %-10d)        */
 87          int     longf   ;    /*  long intである. (%1xか%1d, 等)     */
 88          long    lnum    ;    /*  数値引数を保持する                 */
 89 
 90 #ifdef FLOAT
 91          double  dnum    ;    /*  doubleの引数を保持する             */
 92 #endif
 93 
 94          for(; *format; format++ )
 95          {
 96              if( *format != '%')             /*  変換しない            */
 97              {                               /*  次の文字をプリントする */
 98                  (*out)(*format, o_param);
 99              }
100              else                            /*  %変換の処理           */
101              {
102                  bp          = nbuf  ;
103                  filchar     = ' '   ;
104                  fldwth      = 0     ;
105                  lftjust     = 0     ;
106                  longf       = 0     ;
107                  prec        = 0     ;
108                  slen        = 0     ;
109 
110                  /*  変換文字の前におくことのできる文字を解
111                   *  釈する (%04..., %-10.6..., 等).
112                   *  フィールドの幅の代わりに*があったら幅は
113                   *  引数並びから得る.
114                   */
115 
116                  if( *++format == '-') { ++format ;  ++lftjust;      }
117                  if( *format   == '0') { ++format ;  filchar = '0'; }
118 
119 
120                  if( *format != '*' )
121                          TOINT( format, fldwth );
122                  else
123                  {
124                          format++;
125                          NEXTARG( fldwth, int );
126                  }
127 
128                  if( *format == '.' ) { ++format; TOINT(format, prec);}
129                  if( *format == 'l' ) { ++format; ++longf ;           }
130 
131 
132                  /*
133                   *   今まですべての修飾文字を取り上げ, *formatは実際の
134                   *   変換文字を調べている. 適切な大きさの引数をスタッ
135                   *   クから取り出して, ポインタ (ap) を次の引数をさし示
136                   *   すように更新する.
137                   *
138                   */
```

(図9·7つづく)

```
139
140                         switch( *format )
141                         {
142                         default:    *bp++ = *format ;
143                                     break;
144
145                         case 'c':   NEXTARG( *bp++, int );
146                                     break;
147
148                         case 's':   NEXTARG( bp, char *);
149                                     break;
150 #ifdef FLOAT
151                         case 'f':
152
153                             NEXTARG( dnum, double );
154                             if( dnum < 0   &&  filchar == '0')
155                             {
156                                     /*  0で空白をうめてしまうのならば,
157                                      * “0000−5.2.”にしないように, − を
158                                      * プリントしなければならない.
159                                      */
160
161                                     (*out)( '-' ,o_param) ;
162                                     --fldwth;
163                                     dnum = -dnum ;
164                             }
165
166                             bp = dtos( dnum, bp, prec ? prec : 6 );
167                             break;
168 #endif
169                         case 'd':   base = 10 ;
170                                     goto pnum ;
171
172                         case 'x':   base = 16 ;
173                                     goto pnum ;
174
175                         case 'b':   base =  2 ;
176                                     goto pnum ;
177
178                         case 'o':   base =  8 ;
179                                     /*次の文へ抜け落ちる*/
180 pnum:
181                             /*  long や int の大きさの引数をスタックから適切
182                              *  に取り出す. 取り出された数が基数 10 の int でな
183                              *  かったら, 符号拡張が起きないように上位ビット
184                              *  をマスクする.
185                              */
186
187                             if( longf )
188                                     NEXTARG( lnum, long )        /* No ; */
189                             else
190                             {
191                                     NEXTARG( lnum, int  );
192                                     if( base != 10 )
193                                             lnum &= INTMASK;
194                             }
195
196                             if( lnum < 0L  &&  base == 10  && filchar == '0' )
197                             {
198                                     /*  “000−123.”とならないように, − をプリント
199                                      *  する. 10 進数だけが − 符号を得る.
200                                      */
201
202                                     (*out)( '-' ,o_param) ;
203                                     --fldwth ;
204                                     lnum = -lnum ;
205                             }
206
```

(図 9·7 つづく)

```
207                     bp   = ltos( lnum, bp, base );
208                     break;
209                 }
210
211
212                 /*  必要ならば文字列を終わりにして文字列の長
213                  *  さ（slen）を計算する．bp は出力する文字列
214                  *  の始めをさし示す．
215                  */
216
217                 if (*format != 's')
218                 {
219                     *bp  = '\0';
220                     slen = bp - nbuf;
221                     bp   = nbuf;
222                 }
223                 else
224                 {
225                     slen = strlen(bp);
226                     if( prec  &&  slen > prec )
227                             slen = prec;
228                 }
229
230
231                 /*  Adjust fldwth to be the amount of padding we need
232                  *  to fill the buffer out to the specified field
233                  *  width. Then print leading padding (if we aren't
234                  *  left justifying), the buffer itself, and any
235                  *  required trailing padding (if we are left
236                  *  justifying).
237                  */
238
239                 if( (fldwth -= slen) < 0 )
240                     fldwth = 0;
241
242                 if (!lftjust)
243                     PAD( fldwth );
244
245                 while( --slen >= 0 )
246                     (*out)(*bp++,o_param);
247
248                 if ( lftjust)
249                     PAD( fldwth );
250
251             }       /* else 終わり                    */
252         }           /* end for(; *format; format++)   */
253 }
254
```

図 9・7 doprnt.c の続き：doprnt( ) 本体

呼びだして文字をプリントする．サブルーチンの残りの部分は，% 変換を行う else 節である．else 節は 100～252 行までである．

102～120 行までのプログラムは，変換の修飾文字を取っておく変数を取り扱う．それらはディフォルト値に初期化された後，必要に応じて変更される．fldwth はフィールドの幅に設定され，fltjust は % 文字に続いてマイナス記号があったら真に設定される．変換 %*… は認められている．ここでは * はフィールドの幅を引数並びから取りだすことを意味している．この方法は

```
printf("%*d", width, x )
```

で printf( ) を呼びだして，x を幅 width でプリントする．しかし，表記 "%＊.＊f" はサポートされていない．精度（小数点の右側）を規定するにははっきり数を書かなければならない．

変換は，140～209 行にある switch で行なわれる．引数はさきほど述べた方法で，NEXTARG マクロを使用して，スタックから取りだされる．%c 引数は int で文字が渡されるので，int としてフェッチされることを思いだしなさい．同様に，%f に対応する引数は double で渡される．192 行目の if は，整数の大きさのオブジェクトが ltos( ) に渡され，その整数が基数 10 でプリントされないときに，符号拡張が起こらないようにするために必要である（ltos( ) については後述する）．

switch から離れるとき，変換は完了している．そして，変換された数を表す ASCII 文字列は bp にさし示される文字列に入れられている．そのバッファは，必要な空白等をつけて，242～249 行でプリントされる．

9.6　構造化プログラミングの考え方

doprnt( ) は，適切な構造化サブルーチンの例ではない．しかし，5 章にある規則から逸脱しているところにはすべて理由がある．まず第 1 に，doprnt( ) は実行速度を上げるために，サブルーチンとしては長い．長いためになってしまった for ループ（94～252 行で終わる）の外側の大きさと else 節の長さ（101～251 行まで）は望ましくない結果である．この else に関係する if は，if 文の近くにもっとも短い動作をおくように構成されている．

```
if( *format == '%' )
```

だったとすると，if 節は数ページの長さになり，それからやっと

```
else
        (*out)( *format, o_param );
```

が現れる．しかし，読むとたぶんその else に関係する条件は何だったのか忘れているだろう．if/else の先頭に短い動作を移動するとこのプログラムは読みやすくなる．

doprnt( ) の実行速度をはやくするプログラムには他にも 2，3 心配な部分がある．第 1 に case 'f' 文の本体（152～168 行）は switch の大きさを減らすためにサブルーチンにするべきである．同様に，case 'o' の本体も先行する全 goto 文

```
255 /*--------------------------------------------------------------*/
256
257 #ifdef DEBUG
258
259 #include <stdio.h>
260
261 printm (format, args)
262 char    *format;
263 char     *args;
264 {
265         /*  printf の変形版のひとつ．デバッグ用に実際の printf を
266          *  使用することができるように printm( ) が呼ばれる．
267          */
268
269         extern int  fputc( );
270
271         DOPRNT( fputc, stdout, format, &args );
272 }
273
274 /*------------------------------------------------------------------*/
275
276 main( )
277 {
278         printm("%d %x %o %ld %lx %lo %3.1f %c %s\n",
279                          0, 1, 2, 3L, 4L, 5L, 6.7, '8', "9" );
280
281         printm("%s %s %c", "hello", "world", '\n'       );
282         printm("should see <string>  : <%6.6s>\n", "string NO NO NO");
283         printm("should see <70000>   : <%ld>\n", 70000L );
284         printm("should see <fffff>   : <%lx>\n",0xfffffL);
285         printm("should see <ffff>    : <%x>\n",     -1  );
286         printm("should see <-1>      : <%ld>\n",    -1L );
287         printm("should see <x>       : <%c>\n",     'x' );
288         printm("should see <a5>      : <%x>\n",    0xa5 );
289         printm("should see <765>     : <%o>\n",    0765 );
290         printm("should see <1010>    : <%b>\n",     0xa );
291         printm("should see <   123>  : <%6d>\n",    123 );
292         printm("should see <   456>  : <%*d>\n",  6,456 );
293         printm("should see <  -123>  : <%6d>\n",   -123 );
294         printm("should see <123   >  : <%-6d>\n",   123 );
295         printm("should see <-123  >  : <%-6d>\n",  -123 );
296         printm("should see <-00123>  : <%06d>\n",  -123 );
297
298 #ifdef FLOAT
299         printm("should see < 1.234>  : <%6.3f>\n",   1.234 );
300         printm("should see <01.234>  : <%06.3f>\n",  1.234 );
301         printm("should see <1.010 >  : <%-6.3f>\n",  1.010 );
302         printm("should see <-1.3  >  : <%-6.1f>\n", -1.26 );
303         printm("should see <  -1.2>  : <%6.1f>\n",  -1.24 );
304         printf("should see <1.234567>: <%f>\n",     1.234567  );
305         printf("should see <1>       : <%1.0f>\n",  1.234567  );
306 #endif
307 }
308
309 #endif
```

図 9・8 doprnt.c の続き：テスト・ルーチン

にかわって呼びだされるサブルーチンにするべきである．いいかえれば，このルーチンが適切な構造であるならば，次のようになっている．

```
do_int( ... , base )
{
        /*  case '0' の以前の内容がここにくる  */
}
```

```
doprnt( ... )
{
        /*    ...    */

        case 'd':   do_int( ..., 10 );   break;
        case 'x':   do_int( ..., 16 );   break;
        case 'b':   do_int( ..., 2  );   break;
        case 'o':   do_int( ..., 8  );   break;

        /*    ...    */
}
```

goto は，サブルーチンのようなマクロに対するよぶんなプログラムを加えないで，サブルーチン呼出しをしないようにすることができる．

doprnt( ) の最後の部分は，DEBUG が #deifne されているときだけコンパイルされるテストルーチンである．このルーチンは図 9·8 に示されている．デバッグ用に，コンパイラと共に供給されている printf( ) を使用できるように，261 行目にある printm( ) を使用する．筆者はいつもライブラリに含まれる予定のファイルに小さなテストプログラムを入れる．このように独立したモジュールとしてそのファイルをコンパイルすることによって，main( ) を含む第 2 のファイルをつくってそれをリンクしなくても，ライブラリ・ルーチンをテストすることができる．DEBUG が #define されていなければ，main( ) はコンパイルされずに，doprnt( ) がライブラリにリンクされる．

9.7　ltos( )

doprnt( ) を完全なものにするためには，あと 2 つのルーチンが必要である．それらは，long を ASCII 文字列に変換する ltos( ) と，double を文字列に変換する dtos( ) である．ltos( ) のプログラムを図 9·9 に示す．

ltos( ) は 3 つの引数が渡される．n は変換される数，buf は変換された文字列がおかれるバッファへのポインタ，base は変換の基数である．base は，2 進数への変換には 2，8 進数には 8，10 進数には 10，16 進数には 16 である．通常使わない基数（基数 7 のような）に変換するのにも ltos( ) を使用できる．しかし，基数の範囲は 2 から 16 である．基数 10 の数は負数でもよくて，この場合はマイナス符号を先頭につけて印刷される．その他の基数ではビット・パターンのようにみえるので，負数として取り扱っても意味がない．

引数 n は，符号つきの数にしてしまいがちだが，unsigned long として宣言されている．この宣言が 36 行目のモジュロ演算の信頼性を高くする．しかし，0 よ

```
 1 char     *ltos( n, buf, base )
 2 unsigned long    n     ;
 3 char             *buf  ;
 4 int              base  ;
 5 {
 6          /*      long を文字列に変換する．16 進，10 進，8 進，2 進でプリントす
 7           *      る．
 8           *      "n"     は変換される数．
 9           *      "buf"   は出力バッファ．
10           *      "base"  は基数（16，10，8，2）．
11           *
12           *      出力文字列はヌル記号で終わる．そのヌル終端記号へのポインタ
13           *      がかえされる．
14           *
15           *      数は，変換されながら，ひとつずつ配列に入れられる．配列には逆
16           *      順に入れられる（例，/0 が最初で，もっとも右のけたがその次に
17           *      入れられる）．リターンする前に場所を逆に入れ換える．
18           *
19           *
20           */
21
22         register char   *bp    = buf;
23         register int    minus = 0;
24         char            *endp;
25
26         if( base == 10 && (long)n < 0 ) /*  数が負数であり，か  */
27         {                               /*  つ基数が 10 ならば， */
28                 minus++ ;               /*  マイナスをつけて正  */
29                 n = -( (long)n );       /*  数にする．          */
30         }
31
32         *bp = '\0' ;                /*  配列には逆順に入れられるか  */
33                                     /*  ら，今のうちにヌルを入れて  */
34                                     /*  おく．                      */
35         do {
36                 *++bp = "0123456789abcdef" [ n % base ];
37                 n /= base;
38
39         } while( n );
40
41         if( minus )
42                 *++bp = '-';
43
44         for( endp = bp; bp > buf ;)     /*  文字列を逆にする  */
45         {
46                 minus  = *bp;           /*  一時的な格納に    */
47                 *bp -- = *buf;          /*  minus を使用する． */
48                 *buf++ = minus;
49         }
50
51         return endp;            /*  終わりのヌルへのポインタをかえす  */
52 }
```

図 9・9 itos.c—long の文字列への変換

り小さいか調べるために 26 行目で通常の（符号つき）long に n をキャストしなければならない．ltos( ) が呼びだされるとき（long から unsigned long に）暗黙の型変換がある．long は符号つきでも符号なしでも同じ大きさだから，この暗黙の変換は望まない影響は及ぼさない．

変換されたけたは逆順にバッファに入れられる（最も下のけたが最初である）．したがって，最後の '/0' が，どの文字よりもさきに文字列に入れられる（32 行目）．

変換が完了したとき，文字列全体の順序を逆にする（44〜49 行）．

実際の変換はどこか変にみえるが，35〜39 行の do ループで行われる．このプログラムは6章で詳しく説明されている．式“0123456789abcdef”はひとつの文字ポインタに評価され，'[ ]' 演算子をどの文字ポインタにも適用することができる．式“0123456789abcdef”[5] は文字 '5' に評価される．ここで，6章で行われた簡単な AND 演算よりも，現在の基数によるモジュラわり算が行われる．（AND は2のべき乗である基数にだけ変換する）．モジュラわり算は演算数の一方が負数のときは予期しない結果になることがある．理由は n が unsigned で宣言されているからである．次の式

```
x == ((x / n) * n) + (x % n)
```

はモジュラ演算では正しいはずであるが，当てにはできない．

9.8　dtos( )

ltos( ) に相当する浮動小数が使えるものが dtos( ) である．これは，double を ASCII 変換する（図 9·10 参照）．dtos( ) で行われる実際の変換作業の多くは，ltos( ) で行われる．だから，dtos( ) の主な機能は double を2つの long に，ひ

```
 1 extern  char    *ltos   (long,  char*,  int      );
 2 extern  double  floor   (double                  );
 3 extern  double  ceil    (double                  );
 4
 5 #define MAXPREC    32                /*  10 進数の小数点の右側におくこと  */
 6                                      /*  のできるけたの最大数.             */
 7                                      /*  n の小数部分を求める.             */
 8 #define FRACT(n) ((n)-(long)(n))     /*                                   */
 9
10 #define min(a,b)   ((a) < (b) ? (a) : (b))
11
12 /*-------------------------------------------------------------*/
13
14 char    *dtos( num, buf, precision )
15 double  num  ;
16 char    *buf ;
17 int     precision ;
18 {
19         /*   double を文字列に変換する. 10 進数の変換だけをサポートする.
20          *   "num" は変換される数（正でなければならない).
21          *   "buf" は出力バッファ.
22          *   "precision" は 10 進数の小数部分のけた数.
23          *           最大精度は 32 である. 精度が 0 のときは、数の整数部分だけが
24          *           変換される.
25          *
26          *   出力文字列はヌルで終わり，そのヌルへのポインタをかえす.
27          *
28          *
29          */
```

（図 9·10 つづく）

```
30
31          static   char      tbuf[MAXPREC+1];
32          int                i   ;
33          char               *bp ;
34          long               lnum;
35
36          precision = min(precision,MAXPREC); /*  必要ならば，精度を切り  */
37                                              /*  捨てる.                 */
38
39          if( num < 0.0 )                /*  数を正に変換し，必要ならば，  */
40          {                              /*  '−'記号をプリントする.      */
41                  *buf++ = '-' ;         /*                                */
42                  num = -num;
43          }
44
45          /*   数の整数部分を取り出して，ltos を使用して ASCII 文字に変換する．そ
46           *   れから，小数部分を分離するためにもとの数から整数部分をひく.
47           *
48           */
49
50          buf  = ltos( lnum = (long)num, buf, 10 );
51          num -= lnum ;
52
53          if( precision )
54          {
55                  /*   小数部分をプリントする．最初，小数部分を数の正数部分に
56                   *   移動するために，小数部分に 10 を精度回かける.
57                   *
58                   */
59
60                  for( i = precision; --i >= 0 ; )
61                          num *= 10.0 ;
62
63                  /*   結果の数に適切に切り上げか，切り捨てを行う．それから，
64                   *   ltos( )を呼ぶときに結果の数の前におく必要なゼロの数を
65                   *   計算する（ltos は数の前のゼロはプリントしない).
66                   *
67                   */
68
69                  num = (FRACT(num)  >=  0.50) ? ceil(num) : floor(num);
70
71                  i = precision - ( ltos((long)num, tbuf, 10) - tbuf);
72
73
74                  /*   最後に小数点，先頭のゼロ，数の小数部分をバッファに転送
75                   *   する.
76                   *
77                   */
78
79                  for( *buf++ = '.';  --i >= 0;  *buf++ = '0')
80                          ;
81
82                  for( bp = tbuf; *bp ; *buf++ = *bp++ )
83                          ;
84          }
85
86          *buf = 0 ;
87          return( buf );
88 }
```

図 9・10 dtos.c-double を文字列に変換する

とつは数の整数部に，もうひとつは小数部に変換する．このやり方の問題点は，数の大きさが long の大きさに制限されることである．当然，long の大きさは double より小さい．つまり 32 ビットの long より，64 ビットの double のほうがより大き

な数を表すことができる．それで，大きな(または，小さな)値を持つ double が dtos( )によって小さくされる．

double の整数部は dtos( )の 50 行目のキャストで取り出される．double を long にキャストするのは，小数を整数に変換することである．変換の一部として小数部が切り上げられる．整数部は 51 行目のひき算で削除される．このひき算の後，整数部はゼロに減らされて小数部はそのまま残される．

n の小数部は 53〜84 行の if 節で取り出される．60，61 行の for ループは，さきに取り出した小数に 10 を精度乗かける(精度は，小数点の右側におきたいビットの数である)．このかけ算は精度のけたを 10 進数の左シフト(小数点を右に移動)と同じ効果がある．興味の対象は，小数部から数の整数部に移っていった．今，残された小数部によってその整数を切り上げるか，きり下げるかしなければならない．これに失敗すると 1.0 のところを 0.99999999 と印刷してしまったりする．この数を丸めるのは，69 行目で行われている．71 行目では 2 つのことを行っている．前と同じキャストと ltos( ) の呼出しを使って，取りだした小数部を文字列に変換している．それと同時に，数を満たすのに必要なゼロの数に i を初期化している(先行するゼロは ltos( )では印刷されないだろう)．最後に，79〜83 行の for ループで，小数点，先行するゼロ，変換された小数部を，もとのバッファにコピーする．

9.9 練　　習

9-1　doprnt( )を使用して，sprintf( )を書きなさい．sprintf( )は，標準出力ではなく文字列に書きだす以外は，printf( )のような動作を行う標準的なライブラリルーチンである．例えば，呼出しは：

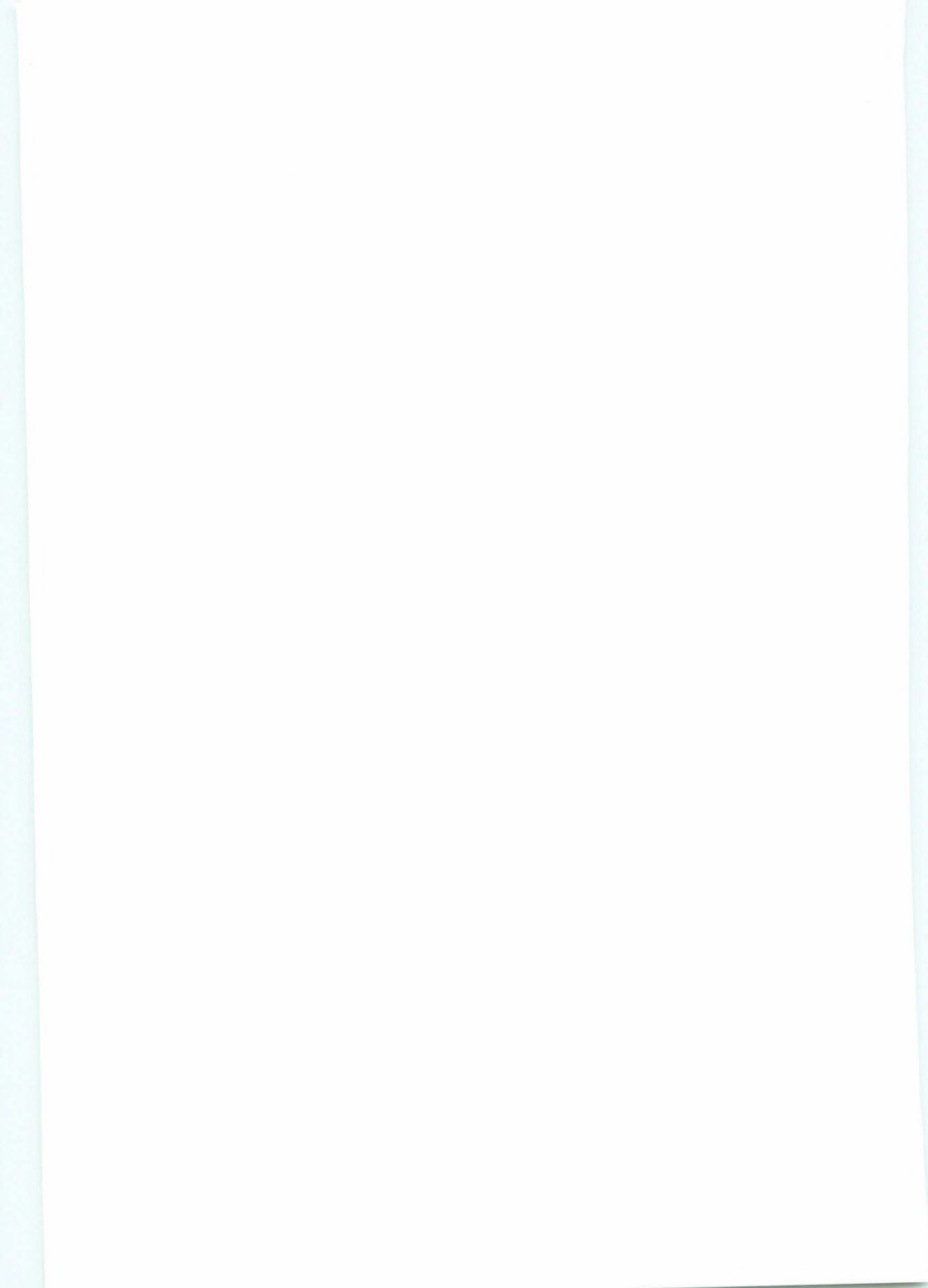
```
char    buf[80];
sprintf( buf, "%d", n );
```

n を ASCII 文字列に変換して，buf にその文字列を入れる(スクリーン上にはださない)．解答が 15 行以上のプログラムになったら，何か重大な誤りがある．

9-2　フィールドの幅を規定する精度部分の代わりに，* でも使えるように doprnt( )を変更しなさい(現在，* は小数点の左にある * しかサポートしていない)．次の決まりも加えなさい．

%p - 16進でポインタの大きさのオブジェクトをプリントする.
%u - 符号なし int を10進でプリントする.
%x - 文字が大文字でプリントされる以外は %x と同じ
%e - double をエンジニアリング表記でプリントする.
[-]m.nnnE[+|-]xx の形を使用する. ここで, n の文字列の長さは精度で規定されており (ディフォルトは6), m は必ず1けたである.

9-3　整数部の値が long で表される double だけではなく, double の値でも印刷できるように doprnt( ) を変更しなさい.

9-4　入力関数 scanf( ) を書きなさい. その関数を規定するのに読者のコンパイラのドキュメンテーションを使用しなさい. 次の変換 %d %c %o %x %f %s だけはサポートする必要がある. * が % に続くときは, 入力列から該当するオブジェクトを取り除き, それをどこにも入れないようにすべきである.

10 デバッギング

本章ではたいへんありふれたデバッグの問題と，C プログラムでのプログラミング・エラーを調べる．そして問題と解決の両方を示す．問題の中には，この本の他の章でもっと詳しく扱っているものもあるが，デバッグの問題をすべて1箇所に集めるために，本章でも再度述べている．

スタックに関係する問題と，ポインタに関係する問題はここでは述べていない．それらは4，6，7章で詳述している．

10.1 デバッグのための printf( ) の使用

たぶん，読者の思い通りになる，もっとも強力なデバッグの手段は printf( ) である．プログラムが適切に動かないときには，プログラムの疑わしいところに，重要な変数の現在の内容や，ループの繰返しの数等がわかるように，printf( ) をたくさん書いておくとよい．通常は，adb や symdeb のようなデバッガを使うよりも，printf( ) 文を挿入するほうが時間がかからない．構造化した方法（5章をみよ）でプログラムを開発中ならば，診断機能を加える前に問題となる位置についてはすでに考えられているはずである．

デバッグに使う printf( ) 文には，デバッグされるサブルーチンの名前や printf ( ) 文がおかれたプログラムの場所を識別できるものを含ませる必要がある．例えば

```
printf("In oz( ), before while(away_the_hours): x = %d\n", x);
```

とする．

＃define の仕組みは，デバッグ文をプログラム中に入れておきたいが，使わな

いときにはそれらは動作しないようにするときに使うことができる（筆者はデバッグ診断は，再びそれを使用する5分前まで消さないようにしている）．これを行う2つの方法が図10·1に示されている．

```
#ifdef DEBUG
#define DIAG(x,y)   fprintf(stderr, x, y);
#define D(x)        x
#else
#define DIAG(x,y)
#define D(x)
#endif

fred()
{
        DIAG("This is only be printed if DEBUG is #defined\n", 0 );
        DIAG("fred(): At top of program, x = %d\n", x);

        D( printf("This will go away when DEBUG isn't #defined\n") );

        for( i = 0; i < it_should_be ; i++ )
        {
                DIAG("fred( ): Doing iteration %d of for loop\n", i );
        }
}
```

図 10・1　診断を削除する2つの方法

DIAGはDEBUGが#defineされていれば，printf( )に展開される．そうでなければ，空の文字列になり，引数は無視される．DIAGを使うとプログラム中に#ifdef DEBUG文を入れて得られるプログラムはきれいになっている．DIAGはマクロだから，第2引数が使われていなくてもマクロには2つの引数を渡さなければならない．しかし，ダミーの第2引数をマクロに渡すこともできる（さきの例の0である）．

図10·1に示されている第2の方法はヌル・マクロDである．DEBUGが#defineされているとき，Dは引数を，この場合はprintf( )文を展開する．前にも述べたように，DEBUGが#defineされていないならば，Dは展開されない．

Dマクロには2，3のただし書きがある．プリプロセッサの中には，これをまったく受け入れないものもある．それで必ずしも移植性はない．同様に，プリプロセッサには，Dへの引数の中にコンマがあると混乱してしまうものもある．コンマには特に注意しなければいけない．最後に，セミコロンの場所も問題を起こす．例をあげて説明する．プログラム

```
if( (2 * b) || !two_b )
        D(  printf("That is the question\n");  )
else
```

```
printf("slings and arrows\n");
```

は，DEBUGが#defineされていなければ，次のように展開する．

```
if( (2 * b) || !two_b )
else
        printf("slings and arrows\n");
```

これはCの文法にかなっていない．

別のデバッグの手段は，サブルーチンが自分のリターン・アドレスを印刷することである．サブルーチンが，呼ばれたアドレスとリンカがつくった非静的なサブルーチンのベース・アドレスの表とを比べて，サブルーチンの実行を追跡することができる．これはスタック・オーバーフローに関係する問題を解析しようとしているときには特に役立つ．その手順は9.1節に説明されている．

printf( )を使った診断の共通の問題は，プログラムの通常の出力と診断の出力が予想できない形で入り混じってしまうことである．これは通常の出力がstdoutに向けられていて，診断のエラー出力もstdoutに向いているときには特に問題である．

MS-DOSとUNIXの出力機構は，この2つの出力の流れを完全に別の出力デバイスとして扱う．したがって，その2つの出力の流れは，必ずしもプログラムからその行が出力されるのと同じ順序でスクリーンにはプリントされない．つまり

```
printf (          "First\n"  );
fprintf( stderr, "Second\n" );
```

は，スクリーン上では

```
Second
First
```

のようにプリントされることもある．行は入れ換わっているけれども，行の中で文字が混ざることはない．

プログラムの中からこの問題を避ける方法は，fprintf( )を呼びだした後で毎回fflush（stderr）かflush（stdout）を呼びだすことである．その他の解法方法もある．すべての出力をプリンタに記録し（SHIFT-PrtScかまたはPを使う），そのとき標準出力を切りかえる（foo > outfileを使う）ことである．標準エラーに送られたテキストは，スクリーンにもプリンタにもプリントされ，標準出力に送られたテキストは，全部指定されたファイルに入るだろう．MS-DOS

は，標準エラーと標準出力両方の出力ストリームの出力装置を，切りかえ可能な標準コマンド・インタプリタ command.com に相当する UNIX のようなシェルがあるが，標準エラーと標準出力両方の装置の切りかえはできない（参考文献参照）.

UNIX では，プログラム >& ファイルは標準エラーと標準出力の両方をファイルに切りかえるのに使われる．UNIX システムでは，stderr と stdout の両方がスクリーンに送られるときは，その2つを再構成できない．しかし，その2つの出力の流れは装置切りかえをするために >& を使うと混ざり合ってしまう（筆者が使用したシステムでは，全部の標準出力がまとめてファイルに送られて，標準エラー出力がその後に続いていた）．次のコマンドを使って，標準エラーと標準出力を別々のファイルに入れることもできる．

```
( プログラム > ファイル1 ) >& ファイル2
```

stdout に送られたすべてのメッセージはファイル1に入れられ，stderr に送られたメッセージはファイル2に入れられる．

10.2　コメントは入れ子にしない

コメントは入れ子にできない．つまり，コメントの内側にコメントを書くことはできない（図10・2 をみよ）.

図 10・2　コメントは入れ子にできない

入れ子になったコメントは，いくつか問題を起こす．簡単にみつけることができる問題は，コメントにしたつもりのテキストをコンパイラが処理しようとして出すエラー・メッセージである．プログラム中でコメントを含んだ部分の外側にコメントをつける必要があるときは，#ifdef 文を使い，それに関係のあるマクロ

名を決して#define しないようにする．

例えば，次のようにする．

```
#ifdef NEVER
        /* NEVER が前に#define されていなければ，
         * この部分のプログラムはコンパイルされない．
         */
#endif
```

NEVER は絶対に#define しない．そして，#ifdef は入れ子にすることができる．

```
#ifdef EMMA

        /* EMMA が#define されているときだけ
         * コンパイルされる．
         */

#ifdef WOODHOUSE

        /* EMMA と WOODHOUSE が#define されているときだけ
         * コンパイルされる．
         */
#endif
        /* EMMA が#define されているときだけ
         * コンパイルされる．
         */
#endif
```

発見しにくいのは，コメントの終わりがないために，コンパイルできないコードである．

```
/* きちんと閉じていないコメント
である．次の while ループはコメン
トの一部になっている． /

while( condition )
        action( );

/* ここは別のコメントである．
コメントの始まりは無視されるが，
コメントの終わりは無視されない． */
```

は，エラー・メッセージを生成せず，while ループをコンパイルしない．

この問題のもっともよい解決法は，コメントの終わりの記号をみつけやすいようにプログラムを書くことである．すなわち，コメントは必ず次の2つのうちのどちらかの方法で書くことである．

```
/*  アスタリスクの列をみていくと
 *  コメントの終わりをみつけるこ
 *  とができる．
 */

/*  コメントの始まりと終わりの記号が          */
```

```
/*   それぞれきちんと縦に並んでいるので,        */
/*   それらをみつけるのは簡単である.            */
```

10.3 lvalue required

もっともよく起こるエラー・メッセージのひとつだが，理解するのがたいへん難しいのは，lvalue required である．問題をもっとわかりやすくするために，lvalue の定義から始める．カーニハンとリッチーの lvalue の定義を次に示す．

> オブジェクトは，メモリの操作単位である．lvalue は，オブジェクトを参照する式である．lvalue のわかりやすい例は識別子である．lvalue を生み出す演算子がある．例えば，E がポインタ型の式であるならば，*E は E がさし示すオブジェクトを参照する lvalue 式である．lvalue という名前は代入式 E1 = E2 に由来する．この式で左（left）の被演算数 E1 が lvalue 式である（注 1）．

lvalue は，式の計算結果が格納される場所である（Pooh（Pooh，Winnie der）が"物を入れる大事なポット"といったように）．つまり lvalue は，アドレス，計算結果が格納される場所である．lvalue は実在する場所—どこかで宣言されている変数か，またはポインタがさし示している場所—でなければならない．

lvalue を完全にする概念は，rvalue である．rvalue は lvalue の値を表す．つまり lvalue が計算結果をおくアドレスであるならば，rvalue はその結果自体である．その数が lvalue に入れられる．もっと正確には，rvalue は名なしの一時的な変数であり，式計算の間に計算結果を一時的に保持するために使用される．この一時的な変数は宣言しないから，直接にアクセスすることはできない．

```
x = (y * z);
```

では（y * z）の結果が，コンパイラによって無名の一時変数に入れられる．この一時的な rvalue の内容は，lvalue x にコピーされる．

```
x = y;
```

では，y の内容が名なしの一時変数にコピーされ，その一時変数が x にコピーされるから，y が rvalue である．この動作はコンパイラに予定していた以上の仕事をさせるという副作用がある．たいていのコンパイラは，この余計なコピーを除

注 1：B. カーニハンと D. リッチー，　C プログラミング言語（Englewood Cliffs, N. J.：Prentice-Hall，1978），p. 183．

く．しかし，y はやはり rvalue である．

rvalue の生命は短い．式の計算が終了すると消されてしまう．いつもではないが，rvalue の内容は式計算の終わりで lvalue にコピーされなければならない．しばらく，この規則の例外を調べてみる．

識別子：	`x = 5`

完全な配列の参照：	`array[i] = 5`
	`array[i+j] = 5`
	`etc.`

構造体のメンバの参照：	`structure.field = 5`

間接的なメンバの参照：	`struct_pointer->field = 5`
	`(struct_pointer + offset)->field = 5`
	`(struct_pointer ++)->field = 5`
	`etc.`

さし示されるオブジェクト：	`*pointer = 5`
	`*(pointer + offset) = 5;`
	`*(pointer++) = 5;`
	`etc.`

図 10・3　lvalue になる式

たくさんの構造体が lvalue を生成する．これを図10・3に示す．ポインタかオフセット計算の一部でなければ，すべての演算子は lvalue でなく rvalue を生成する．そして，その rvalue は lvalue にコピーされなければならない．

```
x++ = 5;
```

で，x＋＋は rvalue（インクリメントされる前の x の値を含む）をつくるけれども，この rvalue は lvalue に格納されないので，"lvalue required" エラー・メッセージを生成する（注2）．一方

```
*++ptr = 5;
```

は，＋＋が rvalue を生成するけれども＊が rvalue を lvalue に移しかえすから誤りではない．

たぶんこの理由は，次の２つの文にコンパイラが行うことを調べてみるともっとよくわかるだろう．たいていのコンパイラは

```
char    *ptr;
*++ptr = 'a' ;
```

注 2：演算子＋＋と－－は状況によって，lvalue か rvalue を生成することがまれにある．

を次のように処理する．

（１） ptr の内容を取りだして，それを一時変数に入れる．

（２） その一時変数をインクリメントして，結果を ptr に入れる．

（３） その一時変数に含まれているアドレスに a を入れる．

この３つの動作は一時変数を含んでおり，ステップ１と２で使用された一時変数は rvalue だが（値を含むから），ステップ３で使用されている一時変数は lvalue である（アドレスを保持し，そのアドレスは計算が終わる前に使用されるから）．ステップ１は，一時変数を初期化する．ステップ２は，一時変数を更新して他にコピーする．ステップ３はポインタ（lvalue）としてその一時変数を使用する．

誤った文

```
int   x   ;
++x = 'a' ;
```

はさきに示したものと同様な処理を行う．

（１） x の内容を取りだして，それを一時変数に入れる．

（２） その一時変数をインクリメントして，結果を x に入れる．

（３） 別の一時変数に a を入れる．

しかし，ステップ３で入れられた一時変数は決して使用されない．結局 a は，（１），（２）とは関係ない一時変数以外のどこにも格納されない．これが，この場合にコンパイラが "lvalue required" のエラー・メッセージをだす理由である．

もうひとつ問題がある．式

```
x = --( y - z );
```

もまた誤りである．これは rvalue[(y − z)] が変更されているが，その値は格納されるので間違いではないように思える．残念ながら，演算子 −− は rvalue ではなく，lvalue に適用されなければならない．この問題は "左手が何をしているのか右手は知らない" というようなことである．y−z を計算するプログラムを生成するコンパイラの部分は結果を一時変数におくが，−− を行うプログラムを生成するコンパイラの部分はその一時変数がどこにあるのかわからない．いいかえれば，−− の処理ルーチンは何をデクリメントするのかプログラマが正確に指示すると思っている．しかし，一時変数がどこにあるのかプログラマにはわからないから，−− の処理部分に教えることはできない．

どの演算子が lvalue を必要とし，どの演算子が rvalue を必要とするのかどう

したらわかるだろうか？　これは，コンパイラで使う文法（カーニハンとリッチーの本の後にある文法など）を調べればよい．lvalue と呼ばれる非終端記号がある．これが8章で文法を詳しく調べた理由のひとつである．読者が文法を必要とするときに，それを使用できるように調べておいたのである．

10.4　演算子に関連したエラー

10.4.1　優先順位のエラー

演算子の優先順位には気をつける必要がある．筆者はよくＣの優先順位図をコピーして，それを使いそうなコンピュータの横にかけている．6章でみたように，＊＋＋p はポインタをインクリメントし，一方＋＋＊p は，p にさし示されているオブジェクトをインクリメントする．

他の演算子，特に代入演算子はさらに油断できない．例えば

```
while( c = getchar( ) != EOF )
        something( c );
```

は

```
while( c = (getchar( ) != EOF) )
        something( c );
```

と評価される．つまり，c は getchar(　) からもどされたのが EOF でなければ，その値は代入された1である．EOF だったら0である．getchar(　) からかえされた文字はなくなってしまう．この問題は次のようにかっこをつけて解決できる．

```
while( (c = getchar( )) != EOF )
        something( c );
```

もっと険悪なのが＋＝演算子とその仲間である．次のようにすることはよくある．

```
while( i++ < 10 )
        do_something( );
```

ここで，1ではなく2を加えたかったらどうすればよいだろうか？　次のようにしてしまいがちである．

```
while( i += 2 < 10 )
        do_something( );
```

しかし，優先順位図では＋＝より＜の方が優先度が高い．したがって，この式にかっこをつけて表すと

```
while( i += (2 < 10) )
        do_something( );
```

である．2が10より大きければ，iに1が加えられる．そうでなければ0が加えられる．2は10より小さいから，iは必ずインクリメントされる．このループは意図していた回数の2倍実行される．これもまた正しくかっこをつければ解決できる．

```
while( (i += 2)  < 10) )
        do_something( );
```

ここでの教訓は注意することである．＋＝は問題を起こしそうだから使うべきではないと考えてはいけない．ただ，使うときによく注意すればよい．

また別の問題がある．次の式を考えてみよう．

```
x = a-----b;
```

は1行に5つのマイナス記号を持っている．かっこをつけるとどうなるだろうか？いくつかの場合が考えられる．

```
x = ( a-- ) - ( --b );
x = ( a-- ) --( -b  );
x = a  -  ( --( --b ));
```

もちろん，最初の文だけが構文として正しい．しかし，演算子が－か－－かの決定は，簡単に実現可能な方法でコンパイラの初期段階（字句解析プログラム）にて行われる．この字句解析プログラムは，Cの構文については何も知らない．意味のあるものが集まるまで，入力から文字を集めるだけである．コンパイラは，たぶんこの3つのうちの中央のものを選び，エラー・メッセージをだすだろう．

10.4.2　評価順序のエラー

C言語では，たいていのプログラミング言語にあることと同様に，同じ優先度を持つ2つの演算子を評価する順序は決まっていない（注3）．これは通常，問題はない．例えば

```
x = --y + --z ;
```

とするとき，yとzをたす前に両方ともデクリメントしている限り，yをさきにデクリメントしてもzがさきでも問題はない．式

```
int      a = 4;

x = --a - --a;
```

注3：この規則の例外は，演算子||と&&である．それは左から右に評価する．評価は左が真か偽か決定されるとき，確実に終了する．

はまた別の問題である．これは言語としては正しい式であるが，たいていのコンパイラは，エラー・メッセージをだす．しかし，左のaがさきにデクリメントされると，式は

```
x = 3 - 2;
```

すなわち1に計算される．もし右のaがさきにデクリメントされていたら，式は

```
x = 2 - 3;
```

すなわち −1と計算される．aはひき算の前に2回デクリメントされる．でも，どちらのaがさきにデクリメントされるかわからない．

式の結合や優先度の順序と，評価の順序を混同してはいけない．結合と優先度は，どの演算子が特定の変数と関係するのか，どのようにみえないかっこを挿入するのかを決定する．評価の順序は，入れ子になっているかっこの同じレベルにある式が評価される順序である．

評価の順序は，サブルーチンの呼出しにも問題を起こす．サブルーチンの引数が評価される順序は決まっていない．次の呼出し

```
a = 4;
foo( a, a++ );
```

には，以前と同様の問題がみられる．左のaがさきに評価されると，foo( ) は foo(4, 4) で呼びだされる．右のaがさきに評価されると，foo(5, 4) で呼びだされる．同様に，呼出し

```
sit( spot( ), fido( ) );
```

では spot( ) と fido( ) のどちらがさきに呼びだされるかわからない．もし2つのサブルーチンがグローバル変数を通して対話したら，ここに問題が起こりうる（ところで，この問題が，できるだけグローバル変数の使用を避けるもうひとつの理由である）．

最後の評価順序の問題は，たし算の演算子である．コンパイラには，たし算は交換法則と結合法則がなり立つ，と仮定しているものがある．したがって，そのようなコンパイラは，＋演算子だけを使った式中のかっこは無視する．例えば

```
z = (a + b) + (c + d)
```

は，たとえ式にかっこがあったとしても

```
z = a + (b + c) + d
```

として扱われるかもしれない．評価の順序を確実にするには次のようにしなければ

ばならない．

```
x =  a + b;
y =  c + d;
z =  x + y;
```

これは通常は問題にならないが，a, b, c, d がサブルーチン呼出しであれば問題になる．

10.4.3　誤った演算子の使用

みてすぐわかる問題は，＝と＝＝，＆と＆＆などの混同である．本当は

```
while( x == y )
```

であるのに

```
while( x = y )
```

とするのには注意すべきである．

後者はyの内容に評価され，前者はxとyが等しいかどうかによって，1か0に評価される．発見することが難しい演算子に関係した問題もある．例えばC言語では，文

```
x + y;
```

だけでもまったく問題はない．それは何の役にもたたないが，多くのコンパイラはこれをみつけても警告メッセージさえださないだろう．これは次のように書くつもりだったのならば問題になる．

```
x += y;
```

同様に

```
x >>= y;
```

とするつもりが

```
x >= y;
```

となっていても，多くのコンパイラは受け入れる．後の文は何もしない，一方さきの文はyビットxを右にシフトする．

ところで，ときおり

```
if( x )
        y++;
```

の代わりに使用されている

```
x && y++ ;
```

のような文をみかける．どちらの文も，x がゼロであれば y はインクリメントされない．しかし後者は読みにくいので避けるべきである．同様に，for 文以外のところでコンマ演算子を使用するのは避けるべきである．かっこを書くのが都合の悪いマクロ定義以外のところで

```
if( condition )
{
        x++;
        y++;
}
```

の代わりに

```
if( condition )
        x++, y++ ;
```

を使用すべきではない．

10.5 制御のながれ

10.5.1 不必要なヌル文，どこにも所属しない { }

C 言語では，セミコロンは文の分離記号（Pascal にあるように）ではなく，文の終了記号である．実際には，セミコロンだけでも言語では正しい文であることを意味する．この文を，何もしないので，ヌル文（空文）と呼ぶ．for ループの本体がないとき等，ときには有効である．例を図 10·4 に示す．

```
strlen( str )                                  /* バイトで文字列の    */
char    *str;                                  /* 長さをかえす        */
{
        register char   *p;

        for( p = str; *p ; p++ )               /* 文字列の終わりに */
                ;                              /* pをすすめる      */

        return( p - str );
}
```

図 10·4 空の本体を持つ for ループ

多くはないが，ヌル文があるとバグになることがある．例えば

```
while( condition );
        --condition;
```

は，"condition がゼロでない間は何もせず（while にヌル文が続いているから），その後 condition はデクリメントされる" といっている．しかし，ループは決して終わらない．同じことが if 文にも起こりうる．

```
if( condition );
        thing_1( );
else
        thing_2( );
```

ここでは，コンパイラが次のようにプログラムを並びかえる．

```
if( condition )
        ; /* 何もしない */

thing_1( );
else
        thing_2( );
```

は，"condition が真ならば何もしない，そして thing_1( ) を行う"．コンパイラが else 節を処理しようとするとき，前にあるはずの if がみつけられなくて，エラー・メッセージをだす．

if の後に '{ }' をおいても，セミコロンがその前にあるときは役にたたない．{ } は while, for, if 等に接していなければならない．例えば，次の記述は正しい．

```
while( condition )
{
        {
                do_something( );
        }
}
```

かっこをこのように使用する理由は，サブルーチンの本体内（先頭ではなく）で宣言される変数の範囲を制限するためである．例

```
alice( )
{
        int     i = 1; <---------- この "i" は下の "i" とは違う変数
                                   である．
        {                                                   |
                int     i; <--------------------------------+

                for( i = MAXVAL; --i >= 0 ; )
                        action();
        }
}
```

ここで，2番目の i は内側の { } 内だけに存在する．それは最初の i とは物理的に異なり，宣言されたときからスタックフレームに加えられ，} があったとき削除される．この状況は，グローバル変数とローカル変数が同じ名前であるときの状態と同じである．ローカル変数がサブルーチンの内側で使用される．しかし，同名のグローバル変数をサブルーチンの内側から直接取りだす方法はない．

一時変数をつくらなくてはならないとき，すでに存在する変数と同じ名前になることを心配したくないならば，次の少しかわった方法が，マクロでは役にたつ．例

```
#define SWAP_INT(a,b) {int temp; temp=(a); (a)=(b); (b)=temp;}
```

は，a と b の内容を交換する．変数 temp は，マクロの { } にはさまれた部分が実行されている間だけ存在する．そのサブルーチン内の他の変数が，temp という名前でも問題はない．マクロの { } の内側の temp は異なる変数である．

10.5.2 誤った if/else の結合

else 文はいつも前にあるもっとも近い if と関係する．例えば，次は誤りである．

```
if( condition )
        if( condition )
                action( );
else <------------------------------ 間違い
        action( );
```

ここでは，誤ったインデントに示されるのとは違って，else は最初の if ではなく，2 番目の if と関係する．この問題は，次のように if/else が続いているときによくみかける．

```
if( ... )
        action( );
else if( ... )
        action( );
else if( ... )
        action( );
```

考えもせずに else 節の中に 2 番目の if 文をおくと，全部まちがった結合になってゆく，このような理由で，if/else を連続させるときは { } を使うべきである．

```
if( ... )
{
        action( );
}
else if( ... )
{
        action( );
}
else if( ... )
{
        action( );
}
```

10.6 マクロ

10.6.1 マクロの優先順位の問題

たいていのコンパイラでは，十分にかっこ付けがされている定数式に対してはコンパイル時に計算が正しく行われている．例えば

```
x = i * (1024 * 6);
```

は，コンパイル時に1 024 ＊6 が計算され，命令コードは最初から x ＝i ＊6 144 だったように生成されるだろう．もしそのかっこが式に含まれていなかったら，コンパイル時の計算は通常保証されない．この場合コンパイラが，かけ算演算子の左から右への結合性に基づいて，みえないかっこを挿入する．つまり

```
i * 1024 * 6
```

はコンパイラによって次のようにかっこ付けされる．

```
((i * 1024) * 6)
```

なぜなら，＊は左から右に結合性があるからである．そのとき，コンパイラは，"式に変数が含まれているので (i ＊1 024) をコンパイル時に計算できない．だから，((i ＊1 024) ＊6) もコンパイル時には計算できない．なぜなら副式 (i ＊1 024) に変数が含まれているから" と一人ごとをいうかもしれない．マクロの内側の組み合わさっている定数を＃define するとき，かっこをつける必要がある．

```
#define ARRAY_SIZE (1024 * 6)
```

これはコンパイラに依存しているから，たとえ読者のコンパイラが正しく計算を行っても，移植性のためにはかっこをつけた方がよい．

マクロの展開で起きるもう少し小さな問題もある．i^2 を計算するマクロ

```
#define SQUARE(i)  i * i
```

を考えてみよ．このマクロは

```
x = SQUARE(a + b);
```

で呼びだされるとき

```
x = a + b * a + b;
```

に展開される．＊は＋より優先度が高いから，コンパイラは式を

```
x = a + (b * a) + b;
```

として計算する．これは期待していたものではない．

プリプロセッサはC 言語を知らない．プリプロセッサは，テキストの置き換え

をするだけである．プリプロセッサは，プログラマがマクロにある引数をどのように計算するのかわからない．指示されたテキストを置き換えるだけである．SQUARE マクロでの問題は，次のように意図的に式にかっこをつけて SQUARE を＃define することで解決できる．

```
#define  SQUARE(i)   ((i) * (i))
```

これで前の例は

```
x = ((a + b) * (a + b));
```

に展開される．これらの問題を避けるには，マクロ内のすべての式に十分にかっこをつけて，マクロの引数もかっこで囲むことである．

10.6.2 期待に反するマクロ引数の置換え

多くの C プログラマは，マクロが引用符（”）で囲まれた文字列は展開されないことを知っている．例えば

```
#define  FOO  xxxx
printf("FOO");
```

では，FOO がプリントされて，xxxx はプリントされない．多くの C プログラマはたいていのコンパイラがマクロ定義の一部である引用符で囲まれた文字列にマクロ引数を展開することを知らない．例えば

```
#define HELLO(name) printf("Hello name\n");
```

のとき，呼出し

```
HELLO( Jean );
```

は

```
printf("Hello Jean\n");
```

に展開される．これは次のようなマクロでは問題を起こすことがある．

この文字列内の展開は，ときには役にたつことがある．例えば

```
PRINT_NUM(name,radix)  fprintf(stderr, "name = %radix\n", name );
```

のとき

```
PRINT_NUM(sarah, x )     /* 16進でプリントする */
PRINT_NUM(bill,  d )     /* 10進でプリントする */
```

で呼びだされたとき，PRINT_NUM は次のように展開される．

```
fprintf(stderr, "sarah = %x\n", sarah );
fprintf(stderr, "bill  = %d\n", bill  );
```

10.6.3　マクロの副作用

マクロは，予期しない方法で変数を変更することがある．マクロ max(x, y) を考えてみよう．これは x か y のどちらが大きいかによって評価する．

```
#define max(x,y)  ((x) > (y) ? (x) : (y))
```

このマクロが

```
y = max( *ptr++, MAXNUM );
```

で呼びだされるとき，次のように展開される．

```
y = ((*ptr++) > (MAXNUM) ? (*ptr++) : (MAXNUM))
```

＊ptr が MAXNUM より大きければ，ptr は 2 回インクリメントされる（＊ptr が，MAXNUM より小さいか同じときには 1 回だけインクリメントされる）．このような動作を副作用と呼ぶ．残念ながら，ctype.h にあるマクロ（isupper, toupper, isdigit）の多くは副作用が起こることがある．それらには注意する必要がある（注 4）．

10.7　16 ビットではない int

10.7.1　精　　度

int，あるいはその他の型が特別な大きさであることを確かめることはできない．筆者はいままで，16 ビットより少ないビットで表されている int をみたことがない．しかし，16 ビットと 32 ビットの int はよくみる．これは機械的な計算での考え方である．int が 32 ビットであると仮定すると，16 ビットの計算機にプログラムを移植しようとするとき，数を切り捨てなければならない．精度が 32 ビット必要ならば，int ではなく long に変数を入れる（int の大きさを計算するサブルーチンが図 5･13 に与えられている）．

注 4：審議中の ANSI 規格の C 言語は，標準ライブラリのマクロ（ctype.h 等）の波及効果を禁じている．しかし，波及効果がないとは思えない．

10.7.2 マクロでの不完全な文

マクロに関係する最後の問題は，if 文を含んでいるマクロである．例えば

```
#define ERR(e,s)    if(e) printf(s)

if( condition )
        ERR( e, "error" )
else
        printf("something else");
```

は期待どおりには動かない．else 文は前にあるもっとも近い if と結びつく．この場合その if は ERR マクロの一部である．上のプログラムは，誤って次のように展開される．

```
if( condition )
{
        if( e )
                printf( "error" );
        else
                printf("something else");
}
```

この問題は，前に問題にしたヌル文を利用して解決できる．C ではセミコロンだけでも正しい文である（何もしないが，誤りではない）．ERR マクロは次のように参照される．

```
#define ERR(e,s)    if(e) printf(s); else
```

これで if は

```
ERR(e,s);
```

で呼びだされるときに，セミコロンひとつが本体である else 節と関係する．

10.7.3 マ　ス　ク

int の大きさは，ワードの上位か下位のビットをマスクするときにも重要である．図 10･5 は，移植性のある形と移植性のない形で表した一般的なマスクの一部である．左の欄は，16 ビットの int を使用していると仮定したマスクを示す．

移植性なし：	移植性あり：	X &= MASK:	X |= MASK:
0xffff	~0	全ビットを 1 にセット	何もしない
0xfffe	~1	最右のビットをクリア	最右以外をセット
0xfff0	~0xf	下位 4 ビットをクリア	下位 4 ビットをセット
0x7fff	((unsigned)(~0) >>1)	最上位のビットをクリア	最上位以外をセット
0x8000	~((unsigned)(~0) >>1)	最上位ビット以外をクリア	最上位をセット

図 10・5　移植性のあるビット・マスク

中央の欄は，int の大きさについて何も仮定していない場合の等価な式である．右の欄は，AND と OR 演算でのマスクの機能を説明している．コンパイラによって異なるが，式は定数だけで構成されているから図10・5の複雑な式のすべてはコンパイル時に評価されるべきである．

10.7.4　定数は int である

単純な定数は int 型である．これは，定数が非整数の引数を期待しているサブルーチンに渡されるときには問題になる．次を考えてみよう．

```
foo( x )
long    x;
{
}

foo( 10 );      /*   誤 り     */
foo( 10L );     /*   正しい    */
```

ここで，foo( ) への最初の呼出しは正しくない．10 は int だから，コンパイラはスタックに int の大きさのオブジェクトとしてプッシュする．しかし，foo( ) は long の大きさのオブジェクトを期待している（この問題は 3 章でも討議した）．foo( ) への 2 番目の呼出しでは，10 に L が続いているので，問題は解決する．L が続く定数は long 型である．

同じような問題は，定数が長すぎても起こる．例えば，16 ビットに含まれることのできるもっとも大きな数は 32 767 である．

```
int x = 40000;
```

とすることができるが，コンパイラはエラーをプリントしない．しかし，10 進の 40 000 は 16 ビットの符号付き数では表せない（符号なしの 16 ビットでは表せるけれども）．40 000 は 0x9c40 である．最上位ビットがセットされているから，0x9c40 は負数（10 進数−25536）である．式 x ＝ 40 000 は x を −25 536 に初期化する．40 000L としても，代入するときやはり丸められるから役にはたたない．この問題を解決するたったひとつの方法は，x を unsigned か long で宣言することである．

10.8　自動型変換の問題

自動型変換は式が計算されるごとに起こる．特に，char はすべて int に変換される．全部の float が必ず long に変換される．したがって，式中の数はその中の

もっとも大きな型に変換される．例えば，式が int と long を含んでいるとすると，int は使用される前に long に変換される．式が long と double を含んでいるとすると，long は使用される前に double に変換される．一方の非演算数が int でもう一方が unsigned であれば，int は unsigned に変換される．変換は，式を評価しながら，ひとつまたは2つの演算数に同時に行われる．

return 文は式を引数とするから，char の大きさのオブジェクトをかえすことはできない．自動型変換は，return が実行される前に char を int に変換する．同様に，サブルーチンへの引数はすべて式である．それで char や float は，サブルーチンが呼びだされる前に int や double に変換されるから，渡すことはできない（注5）．

式の一部とせずに変数を使用することはできないから，型変換はすべての式で行われる．変数を char や int で格納してメモリを節約しない方がよい（メモリが本当にわずかしかなかったり，配列を使用しているのでなければ）．つまり，char の配列には意味がある．たったひとつの char には意味がない．int を使用すればよい．メモリの節約は，プログラムが型変換を必要としているので錯覚を起こさせる．数を int ではなく char で格納すると，データ領域のメモリを1バイト節約できるかもしれないが，同時に型変換のために20バイトも使う．2つの double をかけ合わせるよりも，2つの float をかけ合わせるほうが時間がかかる（なぜなら float は使用される前に double に変換されるから）．

自動型変換のために起こる問題が他にもある．ループ中の比較

```
unsigned x;

for( x = VAL; x > -1 ; --x )
        action( );
```

は x の値に関係なく，いつも偽である．ここで−1（ビット・パターンは 0xffff）は符号なしの数のように扱われる．−1 は int であるが，自動型変換が unsigned にかえる．0xffff は値 65 535，16 ビットの unsigned int で表すことができるもっとも大きい数，である．x はもっとも大きな数より大きくなることはできないためこの比較はつねに偽になる．

32 ビットの long の下位 16 ビット以外をすべてをクリアするために

```
long    x;
```

注 5：ひき数を char であると宣言することができる．しかし，これは通常効力はない．

```
x &= 0xffff;
```

を使うことはできない．0xffff は int で，しかも負数である（値は −1）．x は long だから，定数 0xffff(−1) は使用される前に long に変換される．32 ビットの数に変換されるとき，−1 は値 0xffffffff になる．したがって，上の式は次のように扱われる．

```
x &= 0xffffffff;
```

式は何もしない．問題は 0xffffL と書けば解決することができる．

もうひとつ型変換の問題がある．式は 2 つの項を同時に評価され，そして，中間結果を 2 つの項と同じ型を持つ名なしの一時変数に入れる．さらに，自動型変換の規則が，この 2 つの項にも適用される．例えば

```
int     a = 10, b = 20 ;

d = a * b ;
```

で，a と b がかけ合わされ，結果は名なしの一時変数に入れられる．a と b は両方とも int だから，型変換は必要ない．一時変数の内容は d にコピーされる．今度も名なしの一時変数と d は int であるから，型変換は必要ではない．式がもっと複雑であるとしたら，より多くの一時変数ができるか，その一時変数がより複雑な使われかたをするだろう．

次の場合を考えてみよう．

```
int     a = 32767, b = 10 ;
long    d;

d = a * b ;
```

数 32 767 は 16 ビットで表すことができる．327 670 は表すことができない．そのため結果を long に入れることにする．しかし，名なしの一時変数の型は非演算数の型と同じだから，式は正しく計算されない．つまり，a と b は両方とも int であるから，型変換は行われない．コンパイラは，a と b の内容が実行時にどうなるのか，コンパイル時にはわからない．わかっていることは非演算数の型だけである．したがって，32 767 は 10 をかけられるだろうが，その結果は int の大きさの一時変数には入らない．327 670(0x4fff6) は int には入らないから，−10(0xfff6) に切り下げられる．コンパイラは，この一時変数と d の型を調べる．d は long だから，型変換が一時変数に行われ（long に変換される），−10 が d に代入される．

問題は演算数の一方または両方に型変換をすることで解決できる．だから，名なしの一時変数の型もかわる．これはキャストで行う．

```
d = (long)a * (long)b ;
```

この例ではキャストはひとつだけ必要であるが，計算がすべて正しく行われることを確実にするために，複雑な式では全演算数にキャストを行うことはよい考えである．C では計算順序は保証されていないし，式を計算しながら型変換が行われることをおぼえておく必要がある．

10.9 extern 文の欠如，または暗黙の extern 文

その他の型に関係する問題は，外部サブルーチンの返り値である．コンパイラは，サブルーチンの明示された宣言（実際のサブルーチンの定義か先行する extern 文）をみつけられなかったら，サブルーチンは int をかえすと仮定する．

この仮定は，サブルーチンが宣言される前にファイル中で使用されたとき，問題が起こることがある．コンパイラは知らないサブルーチンをみつけると，シンボル・テーブルにそのルーチンを登録し，その返り値は int であると示す．コンパイラがその後で実際の定義をみつけ，その定義が int 以外のものをかえすことになっていたら，コンパイラは同じ名前を持っているが返り値が異なる 2 つのサブルーチンが宣言されていると思う．コンパイラは "type mismatch"（型が合わない）というエラー・メッセージをだす．

C はこの問題―extern 記憶クラス―が起きないようにする手段を与えている．extern は実際には "external"（外部の）の意味ではない．むしろ，変数（またはサブルーチン）名を特別な型を持つものとしてシンボルテーブルに入れるための，コンパイラへの命令である．コンパイラは，リンカにその変数（またはサブルーチン）のメモリでの実際の場所（コンパイル時ではなく，実行時の）を探させる．いいかえれば，extern は宣言であって，定義ではない．変数宣言は，コンパイラにシンボル・テーブルに要素を入れるように教えるだけである．定義は実際に変数のメモリを確保させ，シンボル・テーブルへの登録も行う．

サブルーチンの外部宣言は次の形をとる．

```
long    john_silver( );  /* long をかえす              */
char    *capn_flint( );  /* char へのポインタをかえす  */
double  billy_bones( );  /* double をかえす            */
```

筆者はたいてい extern 文を全部ファイルの先頭に集める．

extern 文の欠如のために起こるその他の問題は，返り値の切捨てである．あるサブルーチンが，実際にはlong かポインタをかえすが，コンパイラがint をかえすと思っているとすると，返り値は使用される前にint の大きさにされる．この切捨てを防ぐために，int 以外の型をかえすように宣言されたextern 文の，サブルーチンを最初に使用する前におく．

extern 文の代わりにキャストを使用してはいけない．この禁止の理由は次のプログラムにある．

```
char    *ptr;
ptr = (char *) malloc( ARRAY_SIZE );
```

malloc( ) の宣言が使用する前にない．ここでは，malloc( ) がint をかえし（なぜなら，int 以外をかえすことを指示するextern 文がないから），そしてかえされたint が使用される前に文字ポインタに変換されるべきであるとコンパイラに教えている．ポインタが32 ビットで，int が16 ビットであれば，問題が生じる．コンパイラは，malloc( ) がint をかえすと思っているから，malloc( ) の返り値を16 ビットに切り捨てる．それからコンパイラはキャストを処理し，切り捨てられた数を32 ビットのポインタに変換する．しかし，その数には切捨てが行われたから，上位16 ビットは失われている．ポインタへの変換は切り捨てられたビットを再格納しない．代わりに最上位バイトにゼロをうめる．とにかく，そのポインタは意味のあるものをさし示さない．

サブルーチンがlong をかえすとき

```
long            x;
extern  long    foo( );

x = foo( );
```

の代わりに

```
long    x;
x = (long) foo( );        /* foo returns a long */
```

を使おうとすると，同じ問題が起こる．また，extern 文がないので，コンパイラは，foo( ) がint をかえすと仮定する．foo( ) からかえされた値の上位16 ビットは切り捨てられ，それから切り捨てられた数をlong に変換する．ここでは，符号拡張が型変換の一部として行われる．foo( ) の返り値の上位16 ビットが失われるだけでなく，その返り値は負数に変換されることもありうる（下位ワードの

最上位ビットが1であったならば）（注6）.

10.10 I / O

10.10.1 scanf( )

scanf(）は，プログラムが別のプログラムにつくられたファイルを読むのでなければ使わない方がよい．scanf(）は，人間と対話することは意図されていない．scanf(）変換機能を使わなければならないのなら，gets(）かfgets(）で文字列を入力し，それからsscanf(）関数をその文字列から必要な情報を取り出すために使用すべきである．

scanf(）は空白を無視し，改行（'\n'）を空白にする．したがって，入力ファイルの1行につき3つの数が含まれているはずのところに，1行に2つの数しかないとき，scanf(）は次の行に3番目の数を取りにゆく．その行からファイルの終わりまで，正しい入力と同調させることはできない．

scanf(）の全引数はポインタでなければならない．アンパサンド（&）を忘れるとひどい結果になる．

```
scanf("%x", num );
```

ここで，scanf(）はnumの内容を渡され，それを数がおかれるべき場所へのポインタであるかのように取り扱う．numの値が0ならば，scanf(）は変換された数をメモリ番地0に入れる．この例は次のようにするのが正しい．

```
scanf("%x", &num );
```

scanf(）の呼出しと他の入力関数を混ぜて使うとき，特に，scanf(）とgets(）やfgets(）をいっしょに使うときには注意する必要がある．次のプログラムを考えよ．

```
int     num;
char    buf[128];

scanf("%d", &num );        /* 数を得る     */
gets( buf );               /* 文字列を得る   */
```

数，改行，文字列を続けて入力すると，次の動作が行われる．

（1） scanf(）は空白をとばしながら，数字に合うまで，文字を読む．

（2） scanf(）は数を整数に変換しながら，数ではなくなる（改行）まで文字

注6：予約語externは，このような宣言では実際は任意であるが，externを使用することは，よいプログラミングスタイルである．

を読む．

（3）scanf( ) は数でなくなった文字を ungetc( ) を呼びだして入力列にもどし，それからリターンする．

（4）gets( ) が呼びだされる．scanf( ) がもどした改行を読み，文字列の終わりとしてそれを扱い，すぐにリターンする．buf は最初で唯一の文字として '\n' をひとつ含んでいる．

この状態は

```
gets( buf );
num = atoi( buf );
gets( buf );
```

か

```
gets   ( buf );
sscanf( buf, "%d", &num );
gets   ( buf );
```

を使用して訂正することができる．

10.10.2　getc( ) はバッファを持った関数である

getc( ) はバッファを持った入力関数である [getchar( ) もバッファを持っているが，getc( ) を呼ぶマクロである]．これは最初に getc( ) が呼びだされ，入力行全体を読み，その行をバッファに入れることを意味している．それから，getc( ) はバッファ中の最初の文字をかえす．2回目に getc( ) が呼びだされると，バッファに2番目の文字をかえす．getc( ) はバッファに文字がなくなるまでキーボードに文字を取りには行かない．入力キーをひとつたたいても，プログラムが入力関数として getc( ) を使用しているならば，プログラムからの動作をすぐには期待できない．最後にキャリッジ・リターンをつけて，行全体を入力するまでは何も起こらない．

コンパイラの中には第二の "直接" 入力関数—getch( ) と呼ばれる—を供給してこの問題を回避するものもある．その関数は，入力行をバッファに入れない．その他のコンパイラには，キーボードからの入力の操作方法を変更させる関数を供給している．UNIX はこの関数 ioctl( ) を呼ぶ．UNIX はプログラムを実行するまえに実行できる（例えば，シェルスクリプトの中から呼びだせる）システム・コール（stty）も持っている．stty row とタイプして端末機をバッファなしの入力モードにする．stty cooked は端末機をバッファ付き入力モードにもどす．低

次元のI/O関数read( )は入力をバッファに入れない．それで

```
int     c;
read( fileno(stdin), &c, 1 );
```

で一文字得ることができる．

しかし，read( )のこの使い方をサポートしないコンパイラもある．上記は何かが欠けている．自分のオペレーティング・システムと直接対話するルーチンを書かなければならないだろう．

10.10.3 変換されるI/O対変換されないI/O

CP/MとMS-DOSシステムにみられる，関連した問題は，変換を行うI/Oと変換をしないI/Oである．C言語は，行の終わり（'\n'）を表すのに1文字を使う．CP/MとMS-DOSは2文字シーケンスCF-LF（0x0d-0x0a）を使用する（UNIXはLFだけ使用する）．したがって，入力関数は通常CR-LFシーケンスを入力では1文字の '\n' に変換し，出力では '\n' をCR-LFにもどす．この動作は，2進データの読込みでは問題になることがある．たいていのコンパイラはI/O変換を停止する方法がある．もっとも普通の方法は

```
fp = fopen( "file", "r" );
```

の変換読み込みでファイルをオープンし，一方

```
fp = fopen( "file", "rb" );
```

の不変換読み込み，2進モード（binary mode）でファイルをオープンする．キーボードから不変換で入力が必要ならば，直接デバイスとやりとりすることができる．詳しくはコンパイラの使用説明書を調べればよい．

DOSでは，コンソール用に別のファイルポインタをつくることができる．

```
fp = fopen( "con",      "rb" );
fp = fopen( "con:",     "rb" );
fp = fopen( "/dev/con", "rb" );
```

図10·6にこの機構を使用して直接プリンタに書く方法を示す．

UNIXでは，端末機と直接会話するためには

```
fp = fopen( "/dev/ttyNN", "r" );
```

と書かなくてはならない．NNは自分の端末機のttyナンバーで置き換える（whoかwhoamiコマンドを使うと，この数を得られる）．

/devディレクトリは，すべてのハードウエア装置をアクセスするために，UNIX

```
#include <stdio.h>

main( )
{
    FILE    *lpr;

    if( !(lpr = fopen("/dev/prn", "wb")) )
        printf("Can't open printer\n");
    else
        fprintf(lpr, "Quo usque tandem abutere...patientia nostra\n");
}
```

図 10・6　MS-DOS 下のプリンタに直接の書込み

によって使われている．/dev はオペレーティング・システム（自分のコンパイラの I/O ライブラリではない）によって管理されている架空のディレクトリである．その中のファイルは実際は I/O システムにあるデバイスである．

DOS は，ファイル名 con を特別なものとして認識する．con があるディレクトリが重要なのではない．つまり，/foo/con とすることもできる．厳密にいえば，DOS のデバイスに書くときには con：，con，CON，または CON：を使う．コンパイラの中には，小文字は受け入れないものもある．あるいは，小文字を要求するものもある．最後のコロンがなければならないもの（con：のように）もある．そして，やはりコロンが使えないものもある．DOS 自体は /dev/con を受け入れるので，筆者が知るコンパイラはすべてこれを受け入れる．それで，/dev/con は他のものより移植性があるように思われる．

10.11　演算子を除く最適化プログラム

読者が書いたプログラムは，まったく間違っていないときでも，コンパイラが最適化しながら，そのプログラムをごみにかえてしまうことがある．たいていの最適化プログラムに関係する問題は，コンパイルするとき最適化しなければ簡単に解決できる．通常，コマンドのスイッチかコンパイラの最適化のパスを実行しないことで，最適化を行わないようにできる．

いくつか例をあげる．数の最上位ビットをクリア（0 にセット）する次のプログラムをみてみる．

```
unsigned x;
x = (x << 1) >> 1;
```

最適化プログラムは，右シフトが続く左シフトは何もしないのと同じであると

考え，その行を無視する．もちろん，

```
x &= (unsigned)(-1) >> 1;
```

の方がよい．

メモリ・マップドのハードウエアにインターフェースしているときに表れる，もっと現実的な問題がある．それはハードウエアの事象が出力ポートへ書きだすことから始まる．ハードウエアは，何をポートに書いたのか関知しない．ただアドレスされるポートを見張っているだけである．つまり

```
hardware.register = 1;

for( i = DELAY; --i >= 0 )
    ;

hardware.register = 1;
```

のような状況で，最適化プログラムはhardware.registerに1が書かれたことをみることができる．最適化プログラムが2度目の書き込みを処理するとき，その間に他の命令によってhardwear.registerを変更していないことに気づく．それで，最適化プログラムは，その2度目の命令を省いて最適化する（2度目の命令は実行されない）．関連問題がある．

```
c1 = uart.data
c2 = uart.data
```

これは最適化プログラムによって次のように変更される．

```
temp = uart.data
c1 = temp;
c2 = temp;
```

同様に

```
uart.data = c;
uart.data = e;
```

のとき，最適化プログラムは，uart.dataが最初に設定されてから2度目に設定されるまで使われないので，最初の代入を省略する．この問題はアナログ／デジタル変換が使用されているときよく起こる．変換処理を始めるためにコンバータに書き込む．それから直ちに変換結果を読む．

```
char *atod = (char *) 0x1000;

*atod = 1;
c = *atod;
```

＊atodに1を書くと変換が始まる．次の文は結果を読む．＊atodは2つの文の

間で変更されたり使用されていないから，最適化プログラムがさきのプログラムを次のようにかえてしまう．

```
*atod = 1;
c     = 1;
```

10.12 練　　習

10-1 max(x, y) マクロと図10·3の2つのマクロを書きかえて，副作用が起きないようにせよ．ただしこれらのマクロを，サブルーチンにかえてはいけない．

10-2 次のプログラムで悪いところはどこか？

```
#include <stdio.h>
#define PRINT(d)        printf("Got %d lines\n", d );

main( )
{
        char    *buf;
        int     c, count ;

        for(count = 0 ; c ; count++);
        {
                if( gets(buf) != EOF )
                        if( c = *buf == 0 )
                                printf("Got <%s>\n", buf );
                else
                        PRINT( count );
        }
}
```

10-3 次のプログラムは何をプリントするのか？　それはなぜか？

```
#define islower(c)      ('a' <= (c) && (c) <= 'z')
#define toupper(c)      (islower(c) ? (c) - ('a'-'A') : (c))

main( )
{
        char    *p = "masonic dozens DEIfy forelock too ..";

        while( *p )
                printf("%c", toupper(*p++) );
}
```

参　考　文　献

参考書と教科書

C 言語の権威のある本は

Brian Kernighan and Dennis Ritchie. **The C Programming Language.** Englewood Cliffs, N. J.：Prentice-Hall, 1978.

である．通常 K & R と呼ばれるこの本は，経験を積んだプログラマにとっては絶対的である．簡潔で，説明の重複がない．カーニハンとリッチーは，はっきりした短い文章で言語全体を取り扱っている．私個人としてはとても好きな書き方である．読者は次の本で K & R を補うとよい．

Samuel P. Harbison and Guy L. Steele Jr. **C：A Reference Manual.** Englewood Cliffs, N. J.：Prentice-Hall, 1984.

この本は，私が知っている限りもっともよい C 言語の参考書である．その内容は広範囲である．最近追加されたものも多く含まれていて，言語は十分に記述されている．しかし，この本は教科書ではなく，参考書である．C 言語の完全で正確な記述は審議中の ANSI 規格（X3J11）に基づいている．この本が書かれているときは実用段階の草案だった．そのコピーは，X3 Secretariat：Computer and Business Equipment Manufactures Association, 311 First Street N. W., Suite 500, Washington, DC 20001-2178 から入手できる．

また，持っているとよい本は

The Unix Programmer's Manual (Revised and Expanded Version). New York：Holt, Rinehart & Winston, 1979.

第 1 巻は，UNIX，バージョン 7 の標準ライブラリにあるサブルーチン全部のド

キュメンテーションをのせている．自分が持っているコンパイラのドキュメンテーションと比較して，自分のコンパイラがどの程度標準的なものなのか調べるのに役にたつ．もし大学の近くに住んでいたら，もっと最新の本が読める．また大学図書館か，コンピュータ・サイエンス学科に行ってみるのもよい．

中級程度の読者にはK & Rよりも役にたつ数冊のC言語の入門書がある．私が好きな本は

Bryan Costales. **C from A to Z.** Englewood Cliffs, N. J.：Prentice-Hall, 1985.

この本は読みやすくて，言語についての論議が完全である．もう1冊良書をあげる．

Alan R. Feuer. **The C Trainer.** Englewood Cliffs, N. J.：Prentice-Hall, 1986.

この本は，Cインタプリタとともに使用することになっている．対話的な方法でC言語を教えている．プログラムをいくつか書いて，それがどのように動くかみる．インタプリタ自体はたいていのコンピュータ用のものが（IBM，マッキントッシュ，他いくつか大きなシェア機種も含めて）手に入る．この本は，多少プログラミングの経験がある人を対象にしており，インタプリタの値段も含めると高額になる．

次に2冊のすぐれた学習用の本をあげる．

Alan R. Feuer. **The C Puzzle Book：Puzzles for the C Programming Language.** Englewood Cliffs, N. J.：Prentice-Hall, 1982.

Clouis L. Tondo and Scott E. Gimple. **The C Answer Book.** Englewood Cliffs. N. J.：Prentice-Hall, 1986.

The C Puzzle Bookは練習問題集で，各問題の詳しい解答がついている．私がC言語を学んでいるときにこの本が出版されていたら，デバッグの時間が数日減っていただろう．練習問題は，あまり知られていないがよく起こるCプログラム中のバグのみつけ方を教えることを意図している．**The C Answer Book**はK & Rでの全問題に対する詳しい解答がのっている．K & Rを読み，その練習問題を解いて得られるC言語の知識以上の知識はないものとしている．

実際のCプログラムの例を調べることも役にたつ．良書を次に示す．

The Dr. Dobb's Toolbook of C. New York：Brady Communications,

1986.
Ted J. Biggerstaff. **Systems Software Tools**. Englewood Cliffs, N. J.：Prentice-Hall, 1986.
Douglas Comer. **Operating System Design, The XINU Approach.** Englewood Cliffs, N. J.：Prentice-Hall, 1984.
William James Hunt. **The C Toolbox：Serous C Programming for the IBM PC**. Reading, Mass：Addison-Wesley, 1985.

The Toolbook は，700ページのCのソース・プログラムを含んでいる．grepの改良型や，完全なライン・エディタとコンパイラも含んでいる．

Dr. Dobb's Journal は一般にCプログラムのかなりよい供給源である．他のコンピュータ雑誌よりも多くのCプログラムを出版している．一年間の予約講読料は29.95ドルである．住所はDr. Dobb's Journal, P. O. Box 27809, San Diego, California. 92128.

System Software Tools と Operating System Design は，両方ともマルチタスクのオペレーティングシステム用の完全なソースプログラムを含んでいる．Comerは，LSI-11用の完全にUNIXライクなオペレーティングシステムを供給している．ただ，オペレーティングシステム本体だけで，ユーザーインタフェース；またシェルはない．Comerの本は低レベルのディスクドライバがのっている．Biggerstaffは，IBM-PCファミリ上で動くマルチタスクシステムを記述している．ウインドウをサポートし，ユーザインタフェースを与えている．しかし，ディスクをアクセスするためにはMS-DSを使用する．Biggerstaffは，Comerよりも受け入れやすい形でオペレーティング・システム設計の基礎を説明している．しかし，Comerは低レベルのディスクインターフェースが詳しくてよい．

Crafting C Tools は，（アセンブラではなく）C言語での低レベルのIBM PCのプログラミングについて書いてある．それは完全で役にたつサブルーチンが多い．

The C Toolbox は，タイトルに抵抗があって，IBM PCは少ししか取りあげない．アプリケーションのプログラミングについて多く述べている．BTREEデータ・ベース管理プログラムのような重要なプログラムをいくつか含んでいる．

数学とコンピュータ・サイエンス

すべてのプログラマは，ある程度の数学の知識が必要である．ブール代数や基礎

的な確率やグラフ理論，等．これらのよい入門書をあげておく．

Romualdas Skvarcius and William Robinson. **Discrete Mathematics with Computer Sience Apprecations**. Menlo Park, Calef.：Benjamin/Cummings，1986.

構造化プログラミングは次の本で説明されている．

Kirk Hansen. **Data Structured Program Design**. Englewood Cliffs, N. J.：Prentice-Hall，1986.

Brian Kernighan and P. J. Plauger. **The Elements of Programming Style**. Englewood Cliffs, N. J.：Prentice-Hall，1978.

Hansenの本はプログラム設計のWarnier/Orrメソドロジーについて述べている．The Elements of Programming Styleは，FORTRANからCにかえて構造化された設計の良い入門書が必要な人にはとても役にたつ．例がすべてFORTRANで書かれている．

データ構造には2冊良書がある．

Robert L. Kruse. **Data Structures and Program Design**. Englewood Cliffs, N. J.：Prentice-Hall，1984.

Aaron M. Tenenbaun and Moshe J. Augenstein. **Data structures Using Pascal. 2 nd ed**. Englewood Cliffs, N. J.：Prentice-Hall，1986.

Krueの説明はTenenbaunとAugensteinの説明より明確である．しかし，後者の方が完全である．どちらの本も例はすべてPascalで書かれている．

アセンブリ言語

8086は，アセンブリ言語のプログラマの立場からみればひどい計算機であるが，IBM PCの持ち主ならそれを学んでおきたいだろう．幸いなことに，C言語でプログラムをつくっていればあまり8086のアセンブラを使う必要はない．しかし，必要になったら次の本で調べるとよい．

John Angermeyer and Kevin Jager. **MS-DOS Developer's Guide**. Indianapolis，Ind.：Howard W. Sams & Co.，1986.

Robert Lafore. **Assembly Language Primer for IBM PC and XT**. New York：Plume/Waite，1984.

Christopher Morgan. **Bluebook of Assembly Routines for IBM PC &**

XT. New York：Plume/Waite, 1984.

MS-DOS Developer's Guide は実際はIBM PCのシステムプログラミングについて述べている．しかし，アセンブラの具体的な使用方法について，実際的でよい例がたくさんのっている．読みやすい，完全な本である．Morganは，小さな8086のアセンブリ言語サブルーチンの本である．アセンブラについては教えていないが，よい例がのっている．IBMからアセンブラを買うと，アセンブラと一緒に言語の参考図書がついている．一方で，マイクロソフトのマクロ・アセンブラはよいアセンブラである（マイクロソフトは通常IBMよりすすんでいる）．しかし，マイクロソフトのアセンブラには，参考図書がついていない．そのためのよい参考書は，

Russell Rector and George Alexy. **The 8086 Book.** Berkeley：Osborne/McGraw-Hill, 1980.

この本からは8086のアセンブラは学べず，参考図書以上のものではない．Z80プログラマのためには

Daniel N. Ozick, **Structured Assembly Language Programming for the Z80.** Hasbrouck Heights, N. J.：Hayden Book Co., 1985.

がある．言語についてだけでなく，アセンブラを読みやすく維持管理可能な形式で書く方法も教えている．

コンパイラの設計と構成

C言語の簡単な文法は，K & Rの214〜219ページにある．もっとよい文法（実際のコンパイラで使用可能な文法）は，HarbisonとSteeleの本にのっている．もちろん，ANSI規格の文法もある．表を使った構文解析テクニックの短くよい説明は

Henry A. Seymour, "An Introduction to Parsing", Dr. Dobb's Journal, 98 (December, 1984) pp. 78-86.

にある．もっと深くこのテーマを調べ，一般的なコンパイラ設計を調べるには，

Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. **Compilers：Principles, Techniques and Tools.** Reading, Mass.：Addison-Wesley, 1986.

がよい．この本は表紙にドラゴンが書いてあったために"ドラゴンブック"と呼ばれた本（Principles of Compieler Design）を書き直したものである．不幸な

ことに，たいていの数学書と同様に，あまり読みやすくはない．不必要に複雑な形で単純な概念が説明されていて，実用的なプログラム例はほとんどない．読みやすいが理解が難しい本は

P. M. Lewies, D. J. Rosenkrantz and R. E. Stearns. **Compiler Design Theory**. Reading Mass.：Addison-Wesley, 1976.

である．最後に全体的に実用的な本は

Per Brinch Hansen. **Brinch Hansen on Pascal Compilers**. Englewood Cliffs, N. J.：Prentice-Hall, 1985.

である．YACC と LEX は，文法とトークンの記述を入力とし，構文解析プログラムと字句解析プログラム用の C 言語のソースプログラムを出力とする．私が知っている YACC と LEX の使用法のもっともよい記述は

Axel T. Schreiner and H. George Friedman, Jr. **Introduction to Compiler Construction with UNIX**. Englewood Cliffs, N. J.：Prentice-Hall, 1985.

である．コンパイラに関係する話題は，ルートまたはスタータップ・モジュールである．UNIX ライクのリディレクションやパイプ，ワイルドカード拡張のサポートをするために，Aztec CII を改造する例がのっている本を次に示す．

Allen Holub, "C-Chest", Dr. Dobb's Journal of Software Tools, 101 (March, 1985) pp. 10-28.

ツ　ー　ル

役にたつ UNIX ツールやプログラムが IBM PC 用にもある．

SH は，MS-DOS で動作する UNIX C シェルをスケールダウンしたバージョンである．これは Dr. Dobb's Catalog (M & T Publishing, 501 Galveston Drive, Redwood City, Calif. 94063 (8001528-6050) からでている．grep のバージョンのひとつを含んでいる UNIX ライクのユーティリティ (/util)・パッケージは前と同じところから出版されている．どちらのパッケージも，ディスクで実行形式のプログラムとそのソースプログラムがいっしょにくる．それらは大変役にたつプログラムで，実際的な大きさの C プログラムのよい例でもある．M & T は **Dr. Dobb's Toolbox** にあるプログラムのいくつかをマシン・リーダブルのバージョンにしてだしている．(小さな C コンパイラとアセンブラ，テキスト処

理のツール，等）すべて完全なＣソースプログラムが含まれている．

make ユーティリティのバージョンがいろいろPolytron Software（P. O. Box 787, Hillsboro, OR 97123；5031648-8595）とLattice Inc.（P. O. Box 3072, Glen Ellyn, Ill. 60138；3121858-7950）からでている．Polytronバージョンはpolymakeと呼ばれ，かなりUNIXコンパチブルである．Latticeバージョンは LMKと呼ばれ，少し高額だが走行中のメモリは少ししかいらない．他にもmakeのバージョンはコンパイラとともに供給されている（これを書いたときには，Microsoft C，Aztec C，Datalight Cはみなmakeを含んでいた）．飾りのないmakeのソースプログラムが1985年の7月にc-chet（Dr. Dobb's Journal，105, p. 20f）でだされた．

lintのよいバージョンはPC-lintである．これはGimple Software, 3207 Hogath Lane，Collegville，PA，19426；(215)584-4261，から出されている．

コンパイラはしばしば変更されるので，コンパイラについては述べていない．ここで述べてもほとんど意味がなくなってしまうだろう．コンパイラを買いたいのならば最近発行されたコンピュータの技術雑誌を読んでみればよい．Dr. Dobb's Journal，the PC Tech Journal，Computer Language Magazineが出版されている．

付　　録 A

優 先 順 位 図

<table>
<tr><th>Associativity:</th><th>Operator:</th></tr>
<tr><td>left to right</td><td>() [] ->[1] .[2]</td></tr>
<tr><td>right to left</td><td>! ~ ++ -- –[3] (type) *[4] &[5] sizeof</td></tr>
<tr><td>left to right</td><td>*[6] / %</td></tr>
<tr><td>left to right</td><td>–[7] +</td></tr>
<tr><td>left to right</td><td><< >></td></tr>
<tr><td>left to right</td><td>< <= > >=</td></tr>
<tr><td>left to right</td><td>== !=</td></tr>
<tr><td>left to right</td><td>&[8]</td></tr>
<tr><td>left to right</td><td>^</td></tr>
<tr><td>left to right</td><td>&&</td></tr>
<tr><td>left to right</td><td>||</td></tr>
<tr><td>left to right</td><td>?:</td></tr>
<tr><td>left to right</td><td>= += -= /= %= >>= &= etc.</td></tr>
<tr><td>left to right</td><td>,[9]</td></tr>
</table>

注：

1．間接的な（ポインタによる）構造体のメンバ

2．構造体のメンバ

3．1項演算子の－

4．さし示されるオブジェクト

5．アドレス

6. かけ算
7. 2進の－
8. ビット毎のAND
9. カンマ演算子

下の行にある方が優先度が低い.

付　　録 B

シェル・ソート

私は長い間，シェル・ソートはシェルゲーム（貝遊び）にちなんで名づけられたと思っていた．K ＆ R のソート・ルーチンを解いてみようとした人ならば，どうして私がこの結論に達したのか理解できるだろう．実は，そのアルゴリズムの名前は，開発者 Donald Shell に由来する．彼は 1959 年に作成した．シェル・ソートは，ソートの対象を小さく分けて動作する．例えば，8 要素の集合を 2 要素の小集合 4 つに分けて，各小集合内を独立してソートする．最後に 8 要素の 1 集合として格納する．小集合に分ける理論的根拠は，もとの集合のでたらめな順序のそれぞれの部分を，とてもはやく順序よく並べることが可能なことである．小集合の配列は次の段階では，もっとはやくソートが簡単になる．どうして小さな集合に分けるのか？　小集合にある要素を，実際にソートするアルゴリズムはたいしたものではない．ただし，このアルゴリズムの動作は，アルゴリズムを使用している集合が目的の順序に近づくにつれてよくなる．バブル・ソートはこの動作を行う．実際にはシェル・ソートを改良されたバブル・ソートとしてみることができる．

処理過程を示す具体的な例をあげる．8 要素の集合から始めよう．

{9, 2, 1, 7, 3, 8, 5, 4}

これを 4 つの小集合 {9, 3}，{2, 8}，{1, 5}，{7, 4} に分割する．

```
9 2 1 7 3 8 5 4
| | | | | | | |
+--------+ | | |
  +--------+ | |
    +--------+ |
      +--------+
```

それから，各小集合を独立にソートする．

次に4要素の2つの小集合（{3, 1, 9, 5}，{2, 4, 8, 7}）にする．

これを別々にソートすると次のようになる．

```
1 2 3 4 5 7 9 8
| | | | | | | |
+---+---+---+ |
  |   |   |   |
  +---+---+---+
```

最後に，8要素の1つの集合を入力として，ソートする．その結果は

```
1 2 3 4 5 7 8 9
| | | | | | | |
+-+-+-+-+-+-+-+
```

である．

この処理過程を本書にあるプログラム（図7・16）と比較する．もっとも外側の **for** ループ（行11）が，各小集合にいくつの要素があるかを決定する．gap は同じ小集合の2要素間の差である．実際の例では，9と3の最初の小集合の要素間の差は4である．4がgapの初期値になる．この最初のforループ内でgapを2で割る．それが効果的に小集合の要素数を2倍にする．

2番目と3番目の **for** ループ（12，13行）は，実際のソート・ルーチンである．同時に全小集合をソートするので複雑だが，アルゴリズムは基本的にバブル・ソートである．最後の場合（gapが1）だけ扱うようにアルゴリズムを削減し，比較の判断文を逆にして，バブル・ソートをつくる．これでソートがどのように動作するかみることができる．

```
for( i = 1 ; i < argc ; i++ )
        for( j = i - 1; j >= 0 ; --j )
                if( strcmp( argv[j], argv[j+1] > 0)
                        exch( argv[j], argv[j+1] )
```

もっとも内側のループでは，バブル・ソートはソートされていない配列にそって，逆順の隣り合った2要素を探す．その2要素がみつかったら，それらの位置

を交換する．このように順序よく並んでいない要素は，配列中で正しい位置におかれる．比較と交換を束にして行わなければならない（ただし，最悪の場合の全要素が逆順に並んでいるとき，右端の要素を左端までN－1回交換しなければならない)．このもっとも内側のループは，誤った位置にある要素を全部，配列中の正しい位置におくために，十分な時間をかけて実行する必要がある．バブル・ソートは常に$O(N^2)$比較を行う．最悪の場合は，比較回数と同じ回数の交換が必要である．バブル・ソートが遅いのはもっともなことであろう．

もう読者は，どうしてシェル・ソートがバブル・ソートを強力にしたものであるかわかっただろう．バブル・ソートは間違った位置にある要素を取り出し，O(log N）回の交換で配列の前の方に移動させる．シェル・ソートは，要素間のインクリメントでごまかして動作を向上させる．これはもっと生産的なやり方で種々のパスを互いに影響させる．インクリメントは互いに同じである必要はない．2のべき乗は，実際はインクリメントのもっとも悪い選択である．Knuthは魔法の力で1，4，13，40，121，((121＊3)＋1)，…がインクリメントのよい選択枝であると決定した．また彼は1, 3, 7, 15, 31, …, 2^N-1のシーケンスがよく動作するともいっている．インクリメントは後のシーケンスを使用して，シェル・ソートはN要素の配列を$O(N^{1.2})$回でソートする．これはバブル・ソートの$O(N^2)$よりもよい．もし興味があれば，この解析の詳細はDonald E. Kunth，The Art of Computer Programming，Vol. 3.（Reading，Mass.：Addison-Wesley，1973）p. 84f. にある．

索　　　引

ア　行

アーカイブ……16
アセンブラ……58
アセンブリ言語……58
後置きの自動インクリメント付き
　間接アドレッシング……66
アドレッシング・モード……61
暗黙の型変換……42

入れ子構造……20
インクルード・ファイル……5

カ　行

外部オブジェクト……9, 105
角かっこ([　])表記……152
間接アドレッシング……63

構造化プログラミング……109
構文解析ツリー……205
構文解析プログラム……205
コード生成プログラム……206
コンパイラーコンパイラ……207
コンパイル時……81
コンマ演算子……48

サ　行

再　帰……199
再帰的下向き構文解析……207
再配置可能プログラム……105
参　照……10

字句解析フェーズ……1
実行時……81
自動前置きデクリメント付き
　間接アドレッシング……67
指標付きアドレッシング……69
ジャンプ命令……73

ステップワイズ・
　リファインメント……118

正規表現……20
前方参照……119

即値アドレッシング……63

タ　行

代入演算子……48

定義と宣言……10

トークン……205
トークン認識プログラム……205
ドライバ・プログラム……2

ハ　行

配　列……144

バ ス……60
パ ス……1
バッカス記法……207

ビットごとの演算子……45

プリプロセッサ……1,5
プリプロセッサ命令……5
分岐命令……73

ポインタ……140

マ 行

メタキャラクタ……20
メモリ直接アドレッシング……62

ラ 行

ライブラリ……11
ライブラリアン……12

リンクマップ……10

レジスタ直接アドレス……62

ワ 行

割当て……10

英 文

allocation……10
AND マスク……47
ar……16
argv……14

Backus Naur Form……207
BNF……207

C コンパイラ……1
cc……2
cl……2
compile-time……81

dtos( )……242

extern……10

fgets( )……271
for 文……49
for ループの break……103
——の continue 文……103
for(;;)……103
fprintf( )……224

getch( )……272
getchar( )……272
gets( )……271, 272
goto 文……130

if/else……101
ioctl……272

JSR……74

K & R……277

lib……17
libm. a……15
libs. a……15
lint……23
longjmp( )……131
lorder……18
LSB……59
ltos( )……240
lvalue……252
lvalue required……252

make……23

makefile ……24
MSB ……59
msc ……2

OR マスク ……47

printf( ) ……224

ranlib ……19
recursion ……199
recursive descent ……207
reference ……10
RET ……74
root ……14
run-time ……81

scanf( ) ……271
setjmp( ) ……131
sscanf( ) ……271
ssort( ) ……182
stdout ……249
switch ……104, 133

tsort ……18
type mismatch ……269

while ループの break ……102
——の continue 文 ……102
while(1) ……103

XOR マスク ……47

YACC ……207

記　号

& (AND) ……46
~ (NOT) ……46
| (OR) ……46
^ (XOR) ……46
?: ……49
#define ……132
#ifdef DEBUG ……248
>& ……250
/dev ……273

訳 者 略 歴

村上　峰子（むらかみ　みねこ）
昭和60年　九州大学情報工学科卒業
現　　在　横河ユーシステム株式会社システム開発本部第二システム開発部

Cコンパニオン

1989年10月25日　第1版第1刷発行

原著者　Allen I. Holub
訳　者　村　上　峰　子
発行者　株式会社　オ　ー　ム　社
代表者　種　田　則　一
発行所　株式会社　オ　ー　ム　社
郵便番号　101
東京都千代田区神田錦町3-1
振　替　東京6-20018
電　話　03(233)0641(代表)

Printed in Japan

印刷　中央印刷　製本　大進堂
落丁・乱丁本はお取替えいたします

ISBN 4-274-07531-1

カバー印刷：池田印刷